C++ MODEL DEVELOPER (CMD)

USER GUIDE

George W. Snyder
Loretta Painter
Jeffrey W. Hester
Applied Sensors, Guidance, and Electronics Directorate
Aviation and Missile Research, Development, and Engineering Center

George A. Sanders III
System Simulation and Development Directorate
Aviation and Missile Research, Development, and Engineering Center

Ray Sells
Michael Fennell
DESE Research, Inc.
315 Wynn Drive
Huntsville, AL 35805

April 2005

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY	2. REPORT DATE April 2005	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE
C++ Model Developer (CMD) User Guide

5. FUNDING NUMBERS

6. AUTHOR(S)
George W. Snyder, Loretta Painter, Jeffrey W. Hester, George A. Sanders III, Ray Sells, Michael Fennell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Commander, U. S. Army Research, Development and Engineering Command
ATTN: AMSRD-RD-AMR-SG
Redstone Arsenal, AL 35898-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

TR-AMR-SG-05-12

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

A

13. ABSTRACT *(Maximum 200 Words)*

C++ Model Developer (CMD) is an open-source C++ source code based environment for building simulations of systems described by time-based differential equations. The principal design objective behind CMD is to provide a tool to go from mathematical representation to working, extensible C++ code with a minimum amount of effort.

The heart of CMD is a powerful simulation kernel that represents significant technology advances in the application of object-oriented principles to simulation development and design. CMD has been successfully applied for simulation development in the U.S. Army Compact Kinetic Energy Missile Program and is documented here as a technology transfer to offer it to a much larger simulation domain and audience.

This manual is the entry-point for examining and using CMD. For those not necessarily wanting to build a simulation, this guide also illustrates innovative application of object-oriented principles to actually simplify scientific computation.

14. SUBJECT TERMS
C++ Model Developer (CMD), Simulations, Compact Kinetic Energy Missile (CKEM)

15. NUMBER OF PAGES
132

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT **UNCLASSIFIED**	18. SECURITY CLASSIFICATION OF THIS PAGE **UNCLASSIFIED**	19. SECURITY CLASSIFICATION OF ABSTRACT **UNCLASSIFIED**	20. LIMITATION OF ABSTRACT **SAR**

Abstract

C++ Model Developer (CMD) is an open-source C++ source code based environment for building simulations of systems described by time-based differential equations. The principal design objective behind CMD is to provide a tool to go from mathematical representation to working, extensible C++ code with a minimum amount of effort.

The heart of CMD is a powerful simulation kernel that represents significant technology advances in the application of object-oriented principles to simulation development and design. CMD has been successfully applied for simulation development in the U.S. Army Compact Kinetic Energy Missile (CKEM) Program and is documented here as a *technology transfer* to offer it to a much larger simulation domain and audience.

This manual is the entry-point for examining and using CMD. For those not necessarily wanting to build a simulation, this guide also illustrates innovative application of object-oriented principles to actually simplify scientific computation.

Executive Summary

C++ Model Developer (CMD) is an open-source C++ source code based environment for building simulations of systems described by time-based differential equations. CMD is primarily for scientists and engineers who desire to use object-oriented features in developing complex numerical simulations of physical systems and mathematical functions. It is simple enough for students to use as a way of learning C++ and simulation modeling, but at the same time is scaleable from simple models to very complex simulations. It eliminates numerous code writing tasks by allowing the developer to focus on the physical aspects of the model and the type of numerical algorithm being implemented.

The heart of CMD is a powerful simulation kernel that represents significant technology advances in the application of object-oriented principles to simulation development and design. The kernel implements an object-oriented paradigm at all levels: the dynamic states are objects, the models themselves are objects, and the aggregate simulation is an object. Complex system model topologies are easily modeled using a unique "train-of-objects" paradigm. CMD has a powerful runtime dynamic asynchronous scheduling capability that allows seamless mixing of discrete and continuous dynamics models.

CMD has been successfully applied for simulation development in the U.S. Army Compact Kinetic Energy Missile (CKEM) Program. It was recognized that the differential equation modeling capability that was so successfully demonstrated on CKEM is applicable well beyond a particular missile flight simulation. Therefore the program is documented here as a *technology transfer* to offer it to a much larger simulation domain and audience.

This manual is the entry-point for examining and using CMD. The goal is to get the user building and running CMD models as quickly as possible. After working through this manual, users will be prepared to build simulations for desk-top analyses, as well as building CMD simulations for more complex and wider-scale applications including real-time and distributed. This manual can also serve as a ready-reference of functionality for the experienced user.

For those not necessarily wanting to build a simulation, this guide also illustrates innovative application of object-oriented principles to actually simplify scientific computation.

Preface

I wrote some time ago[1], "Simulation recreates aspects of the real world and differential equations are the engines of simulation" and quoted J.M.A. Danby[2], *"A differential equation is like a machine that is capable of movement and ready for action. It only requires some activation: pressing a button, inserting a coin, lighting a fuse - or running a computer. Then it will describe and give details of something that is continually changing. A chemical compound is formed or broken up, a mortgage is repaid, a population increases, a disease spreads, a miss-hit golf ball curves away..."* I then added, "Or a missile flies and is steered on its intended trajectory."

Much like the analogy of a Compact Disk (CD) to a CD player, C++ Model Developer (CMD) is the engine that "plays" differential equations (DEs). A particular set of DEs are the tunes on the CD and CMD is the DE player. As a DE player, CMD leverages capabilities of C++ to mechanize, in code, object-oriented principles that are remarkably powerful for mathematical modeling.

CMD is the culmination of years of experience in using many high-quality digital simulation frameworks (in many computer languages), along with some new ideas along the way. It began as an independent weekend experiment (that stretched into many more weekends) and an idea: a fractal, self-similar at-any-scale, approach to an object-oriented simulation architecture where everything is an object at all levels – the states in the DEs, the DEs themselves, collections of DEs, and even the whole simulation itself. Such an architecture would truly be an embodiment of object-oriented principles. The outcome was uncertain – could such an architecture be mechanized in code? The heart of CMD, an "object-oriented simulation kernel," was the successful result of seemingly almost endless code explorations. Given the ambitious features of the architecture, the resulting kernel was surprisingly compact (less than a couple hundred lines). Over time, the kernel proved itself useful with the application to many tasks, albeit those associated with the developer (the author). Then opportunity came – the U.S. Army Compact Kinetic Energy Missile (CKEM) Program wanted to explore the benefits of C++ simulation and expressed a willingness to consider CMD. Would CMD stand the ultimate test: real-world application outside the hands of the developer? The simulation was quickly embraced and used by the CKEM team to support flight test activities and accomplish a timely simulation deliverable.

It was recognized that the differential equation modeling capability that was so successfully demonstrated on CKEM is applicable well beyond a particular missile flight simulation. Therefore, the program is documented here as a *technology transfer* to offer it to a much larger simulation domain and audience.

---

[1] Sells, H.R. *CSF User's Guide*, U.S. Army Aviation and Missile Command, Systems Simulation and Development Directorate, Redstone Arsenal, AL, (2001).

[2] Danby, J.M.A. *Computer Modeling: From Sports to Spaceflight...From Order to Chaos*, Willmann-Bell, Richmond, VA, (1998).

The C++ language, while very powerful, can be very daunting to use, just due to its sheer bulk and vast capabilities. Thus, the promise of C++'s power is balanced by the challenge to harness it to provide features that truly extend current simulation application capabilities.   The objective in the design of CMD was to transparently leverage this tremendous power "behind the scenes" to the simulationist to make it *easier* to model DEs. CMD aspires to the philosophical goal, "Simple things should be simple and complex things should be possible." Hopefully it marks a significant step toward this ideal.

The end-objective is not building the simulation itself for its own sake but using the simulation to provide answers – this manual describes using CMD to be an efficient means to that end.

Ray Sells
December 18, 2004

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

LIST OF TABLES

1   INTRODUCTION

1.1 Preview

C++ Model Developer (CMD) is an open-source C++ based environment for building simulations of systems described by time-based differential equations. It provides the mechanisms every person needs to develop simulations using the C++ programming language, and is based on man-years of development using proven design patterns.

Who is CMD for? CMD is primarily for scientists and engineers who desire to use Object-Oriented (O-O) features in developing complex numerical simulations of physical systems and mathematical functions. It is simple enough for students to use as a way of learning C++ and simulation modeling, but at the same time is scaleable from simple models to very complex simulations. It eliminates numerous code writing tasks by allowing the developer to focus on the physical aspects of the model and the type of numerical algorithm being implemented.

CMD is unique from other commercial simulation products in that the user works directly in a C++ source code environment to build models. At first this may seem like a disadvantage. After all, graphical-user-interfaces that resemble block diagrams and automated code generators insulate the simulation developer from the complexities of the underlying native code – and some products do this extremely well. Without arguing whether this is indeed always the superior approach to simulation development, CMD approaches simulation development from a completely different angle – provide a highly simplified, but very specific, structure and functionality to describe and simulate a time-based dynamical system. In other words, use the highly flexible, but complex, mechanics of C++ to design a code architecture that gives the simulation developer a high degree of simulation functionality, while at the same time abstracting the complexities and internal operations of these functions inside the underlying code.

The heart of CMD is a powerful O-O simulation kernel that performs all simulation infrastructure functions. The kernel implements an O-O paradigm at all levels: the dynamic states are objects, the models themselves are objects, and the aggregate simulation is an object. The kernel is efficiently written in less than 200 lines of code and can be easily wrapped or embedded for specific applications (real-time for instance).

Above all, the fundamental objective of CMD is the ability to go from mathematical model representation to efficient, working C++ code with the minimum amount of effort.

1.2 Purpose of Manual

This manual is the entry-point for examining and using CMD. The goal is to get you building and running CMD models as quickly as possible. After working through this manual, you will be prepared to build simulations for desk-top analyses, as well as building CMD simulations for more complex and wider-scale applications including real-time and distributed. This manual can also serve as a ready-reference of functionality for the experienced user.

1.3 What Level of Expertise is Required to Use CMD

It is assumed you know something about programming, simulation, and modeling. However, you don't have to be an expert in any of these.

Engineers and scientists by their training are very adept at representing physical systems in terms of mathematical equations and expressions. What is needed is a tool that they can use to compose these models into working simulations without requiring a lot of computer science expertise. CMD attempts to meet this goal.

Three areas of expertise are presumed to use this manual:

1. Representing physical systems as differential equations,
2. Familiarity with O-O methods and principles, and
3. A beginning knowledge of C++ and its syntax

This guide assumes you are comfortable with representing models as differential equations, including state variable representation. An excellent reference is Danby [1]. CMD is an O-O architecture. Don't be too concerned if O-O principles are not totally clear now - it will become apparent how these principles can be applied to simulation as we progress through this guide. A working knowledge of O-O vocabulary would be helpful before proceeding such as knowing that an object is an instantiation of a class. If the preceding sentence was Greek to you, you can consult the first chapters in almost any O-O programming for an overview of O-O methodology. Eckel[2] is a good starting point that explains O-O techniques in the context of C++.

Finally, let it be emphasized that a detailed knowledge of C++ is NOT required to use CMD. In fact just the opposite is true - CMD was designed to leverage the power of C++ while hiding much of its complexity from the user. CMD uses the power of C++ to abstract away from you the inner workings and complexity of the simulation infrastructure, so that you can build and use simulations from a simplified Application Programming Interface (API). Only a knowledge of C++ syntax and cursory idea of how classes work is required to effectively use CMD. The aforementioned reference (Eckel [2]) is a very good starting point for learning C++. Stroustrup[3], who invented C++, is a very good book but may be a little intimidating to new C++ users. It is essentially the "bible" of C++. From a practical point, perhaps it is best to visit the book store and select a book that matches your learning style and level of expertise.

1.4 Feature Overview

Why use CMD? CMD implements many tried-and-tested features to make building simulations easier. In keeping with the philosophy "less is more," each feature emphasizes making things easier.

1.4.1 Open-Source

The CMD O-O simulation kernel is open source with a very liberal usage license. First of all, the source code is part of the distribution. The user can do anything they want to with it including changing it, including it in other software, or even selling it. This open-source usage policy is even less restrictive than most: Code developed using the CMD OSK does not have to be open-source. The only restriction is that you cannot prevent others from using the open-source kernel that is in the CMD distribution.

On a related note, no other software besides a standard C++ compiler is required to use the CMD. Thus, the CMD open-source software requires no proprietary or commercial software for utilization thus simplifying its open-source policy.

The whole intention of the open-source usage policy is to get the source code out among active users and developers and solicit from a diverse community any fixes and improvements.

1.4.2 Compact Object-Oriented Kernel

An O-O Simulation Kernel (OSK) is the core and underlying engine of the CMD. The tightly-coded OSK ($<$200 lines of code) provides a full 100 percent O-O mechanism to service all the required simulation services such as collecting and executing the models in the correct order, sequencing the clock, and controlling the numerical integrator. The key to the OSK's power and flexibility in a little amount of code is that everything is an object: the individual states, the models comprised of states, as well as the entire simulation itself. This pure object-orientation makes it extremely flexible to integrate into and extend the OSK at any level so it can be tailored for application for real-time programming clear up to being an embedded constituent model object in a larger simulation.

1.4.3 Easy "Installation"

CMD, and its underlying OSK, were designed to be copied, compiled, and running on any machine with a C++ compiler in less than a minute. Installation consists of simply copying files - no installer is needed. No customized code or libraries for a particular operating system or platform are needed. CMD is plain vanilla C++ code and has been successfully tested and ran on multiple compilers on multiple platforms (Linux/Unix, Windows, and Macintosh) with absolutely not one line of code changed.

1.4.4 Easy Synchronization of Models

Getting model state variable derivatives and discrete events out of sync with the state variables is perhaps the most difficult issue to address in building simulations. It is the most error-prone area in implementing numerical integration algorithms, especially where discrete and continuous models are present. A well-known and experienced simulationist, Crenshaw [4], says, "As obvious as the point may be, getting x (states) out of sync with t (time) is the most common error in implementations of numerical integration algorithms, and I can't tell you how many times I've seen this mistake made." This mis-synchronization is even made more insidious in that, except for the most simple of systems, it is hard to recognize and discern in the (incorrect) simulation results.

CMD has a powerful *sample* function that allows clear and precise placement of state derivatives for mixing of discrete operations within a continuous model. The *sample* interface enables in-line placement of discrete operations intermingled with the definition of continuous state derivatives. This makes it very easy to code and see where the discrete operations occur with respect to the definition of the continuous state derivatives.

This in-line interface for mixing continuous and discrete models has proved itself "in-the-field" by years of usage in other modeling and simulation environments including National Aeronautics and Space Administration's (NASA) MAVERIC [5] and the Kinetic Energy Anti-Satellite Digital Testbed [6] where it was used to discover mission critical flight software integration bugs[7].

1.4.5 Easy Scheduling of Asynchronous Events

Asynchronous events occur out-of-step with the integrating time-step. Typically, one-time special events occur at these times or it might be desired to simulate a "drifting" clock process where the time between periodic sample instances slowly drifts from a constant value. Dealing with these processes is very difficult where the simulation resolution (i.e. granularity) is constrained to the integrating time-step. The OSK provides a mechanism for easy scheduling of asynchronous events *at any time,* at any granularity regardless of whether the time is an integral multiple of the time-step. Events at any time can be dynamically scheduled before or during runtime. Normal operation of the numerical integration algorithm is not interrupted. The dynamic nature of the scheduling obviates the need for cumbersome adding of event times to a queue (and then having to delete or reschedule the event times if they change).

1.4.6 Multiple Sequences of Objects

CMD features a unique "train-of-objects" architecture for easily specifying the models to be executed during the simulation. For instance, stages on a missile are easily modeled as different sequences of models that are executed during the simulation. The train-of-objects architecture APIs are very flexible: models can be turned off and on during the run and the same models can be reinitialized at the start of a "stage" or continue execution from its state in the previous "stage."

1.4.7 Adding New Numerical Integration Algorithms

CMD leverages the O-O approach by using the inheritance feature for allowing the user to create and use their own numerical algorithms of choice. A base class is furnished with a default fourth-order Runge-Kutta integrator. A new integration algorithm is easily added by inheriting from this base class. The base class already furnishes APIs to access and control the clock and time-stepping so only those features unique to the new integration algorithm need to be added.

1.4.8 Powerful Utility Library

Besides the simulation functionality itself, a key factor affecting simulation usability is the ease-of-use of the supporting libraries. Vector and matrix classes with operator overloading are furnished so that manipulations can be performed using familiar operator symbology. Intuitive table-lookup and input application interfaces are provided. The design philosophy behind these libraries was to leverage the power of C++ for compact, easy-to-use intuitive interfaces while abstracting the underlying complexities from the user.

1.4.9 Platform Independent

CMD was written in standard C++ and utilizes the Standard Template Library and should run "as-is" on any platform that supports these. It has been tested on Windows (NT, 2000, XP), Linux (Red Hat and Mandrake), and Macintosh OS (X) operating systems. CMD uses absolutely no *#ifdef* statements (for compiler-specific instructions) to ensure a high probability that it can be ported to compilers it has not been tested on.

1.5 The Way Forward

The goal of this manual is to enable you to use CMD and leverage its features as quickly as possible. As the name "User Guide" implies, the purpose of this manual is to describe CMD as a working tool to build simulations, not the internals of CMD itself. Towards this end, it is informally written and is designed to get someone familiar with programming, simulation, and math modeling up and running with CMD as quickly as possible. The next section is an example meant to be coded as the section is read to rapidly familiarize you with CMD capabilities.

2   A QUICK-START EXAMPLE

It is the authors' opinion and experience that effective learning by most people can best be achieved by progressing from the concrete to the abstract. Instead of showing an abstract idea first and relying on the reader to extend this to a particular application, the opposite approach is adopted: show a specific example first and then the abstraction to a more general case becomes obvious. In other words, learn by specific example and then, based on that experience, abstracting to the more general is easily accomplished. So, without digging through a lot more documentation, let's illustrate some features by actually building a model – a simulation "Hello World" equivalent. While conceptually simple, the example is complex enough to fully illustrate CMD's architecture and advanced features.

The example also serves to show how the O-O paradigm lends itself very elegantly to time-domain simulation.

If some things aren't clear now – don't worry. They will be fully explained in the following chapters.

2.1 Preview

Perhaps the best way to show what CMD is and how to use it is with an example. In this section, we will build a fully working simulation for a missile pursuit model. As a preview of some of CMD's advanced features, we will simulate a target, missile, and radar all interacting with each other, including changing the simulation model execution sequence. To further emphasize that CMD can be used to simulate any system represented by time-based differential equations, even at the component-level, a simple servo model is developed to point the radar.

The code in this section is meant to be typed and tested as-you-go. Command-line instructions are given to compile and execute the code in both Windows and Linux.

Special Note:
The code for the completed example is included in the distribution as directory ex_0.

2.2 Description of System to be Simulated

Figure 1 illustrates the system to be simulated that consists of three entities: a target, missile, and radar in a single x-y plane.
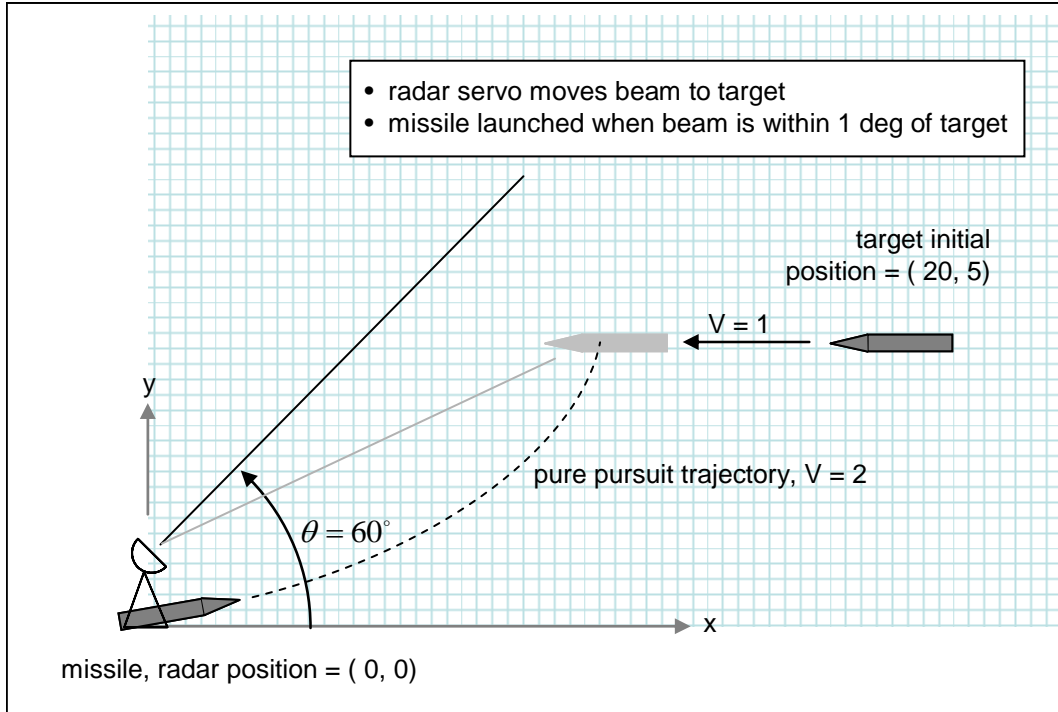
Figure 1. Simple Missile System for Simulation

The target is flying in the direction of the radar and missile which are co-located at the origin. The target is flying at constant velocity and hence its equations of motion are:

$$\frac{dx_t}{dt} = V_x$$

$$\frac{dy_t}{dt} = V_y$$

The radar is initially pointing 60 degrees from the horizontal. A servo-loop attempts to point the radar directly at the target. The closed-loop dynamic model of the servo is shown in *Figure 2*. The servo attempts to null the difference between θ_{radar} and θ_{target}. The closed-loop dynamics are represented by the second-order transfer function shown in the figure.

$$\frac{y}{u} = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$
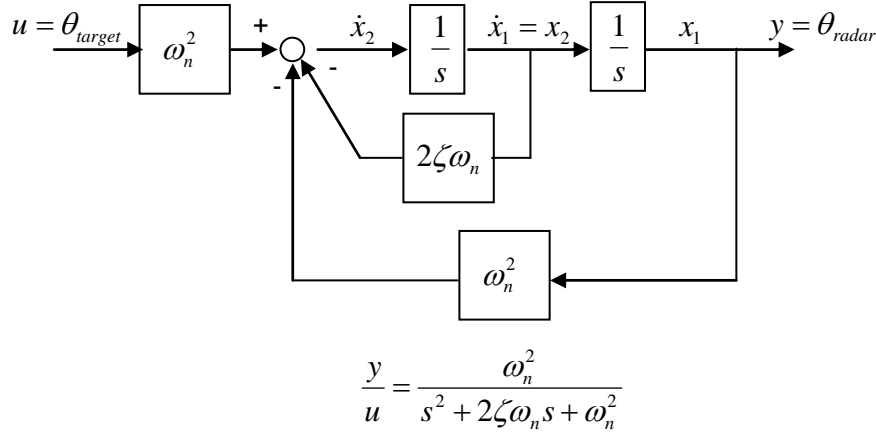
Figure 2. Radar Pointing Servo-Loop

The following state equations for this system are derived directly from the block diagram:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\omega_n^2 & -2\zeta\omega_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \omega_n^2 \end{bmatrix} u$$

Once the pointing error of the radar is reduced to less than one degree, the missile is launched. Like the target, the interceptor missile flies at constant velocity. The missile uses a pure-pursuit strategy for chasing the target. These equations for the missile's velocity vector always point its velocity vector at the target:

$$\frac{dx_m}{dt} = V_m \frac{x_t - x_m}{D}$$

$$\frac{dy_m}{dt} = V_m \frac{y_t - y_m}{D}$$

$$D = \sqrt{(x_t - x_m)^2 + (y_t - y_m)^2}$$

Of course, flying directly at the target is not the most efficient tactic (leading the target is better), but is used here for simplicity of illustration. Again, the missile is launched only after the radar pointing error becomes less than one degree.

System operation terminates when the interceptor missile comes within 0.1 of the target.

Note the complexity of this problem even though the individual models are relatively simple. The model execution sequence changes abruptly during the simulation. At first, only the target and radar models are functioning. Then, at an event triggered by the radar, the missile model is added to the execution sequence. This is not a trivial problem to handle in many simulation environments. In many cases, complex switching and execution logic must be built into the models themselves to address the execution sequence requirements. In addition to the added and error-prone complexity, adding execution logic to the models significantly hinders their reuse for

other situations. CMD, with its O-O kernel, has a unique "train-of-objects" structure to elegantly address complex execution logic without cluttering the models themselves with control code.

The models in the example were also selected to emphasize the scope of modeling that can be done with CMD. System-level characterization and interaction at the macro-level (the constituents of a missile interceptor system) clear down to component-level modeling (a feedback servo) are easily represented in CMD in that these processes are all represented by differential equations.

2.3 Building the Models

We will build a full working simulation of the previously described system step-by-step in this section.

CMD implements the O-O paradigm with a kernel that orchestrates the action of the model objects. Simulation execution consists of the kernel sequentially sending a series of messages to each object, as shown in *Figure 3*. Three kinds of messages are sent by the kernel: initialize, update derivatives, and report results. Each object has its own unique method that performs its appropriate model computation in response to the message. Simulation execution consists of the kernel sending the appropriate sequence of messages to the models and processing the results. The kernel takes care of propagating all the models' states after sending messages to their update methods at each time-step.
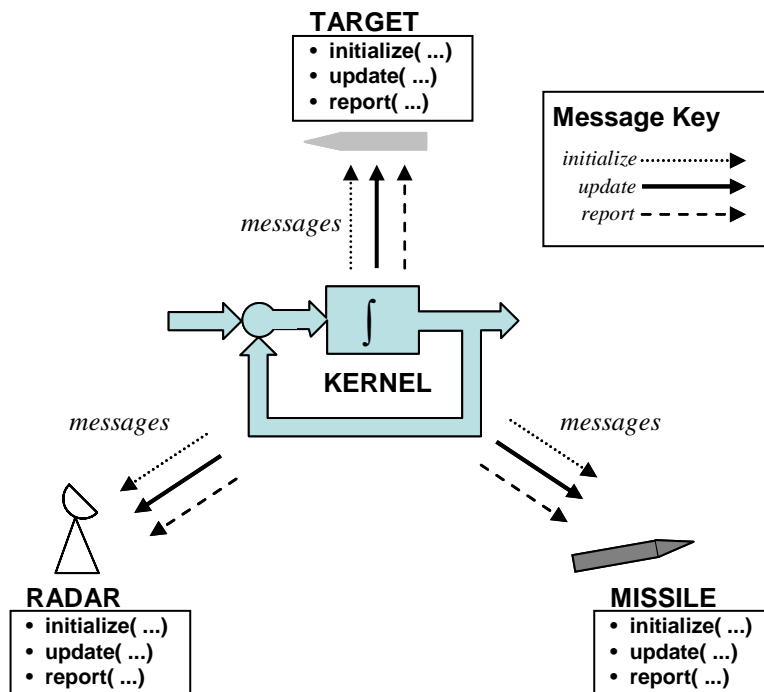
Figure 3. Object-Oriented Kernel Operation

In C++, building a model object consists of overriding methods that are called by the CMD kernel. These methods take care of initialization, defining derivatives, and reporting the models' output. The kernel provides a number of variables to control its operation. In addition to calling the right method in a model at the right time, the kernel also provides a flexible "train-of-objects" organization sequence that can change the sequence of model objects to be executed, if necessary.

After brief instructions for building code from this manual, the following sections detail the code construction C++ classes for the target, missile, and radar models.

2.3.1 Preparation

2.3.1.1   Guide Presentation Style

Most computer programming texts use a mono-spaced font to delineate computer code from the explanatory text. In this guide, we follow Stroustrup's [3] rather unconventional style of listing code in an italicized proportional font. The authors have found that this enhances readability because the code examples can be more compactly represented than with the conventional mono-font.

In the narrative used throughout this guide, all lines of code are numbered followed by a leading tab sequence of dashes. The line numbers are used to reference the code from narrative explanation that accompanies each code snippet. For example:

```
0 --------class Target : public Block {
1 --------  public:
2 --------    Target( double x, double y, double vx, double vy) {
3 --------      this->x = x;
```

Target class is declared[0] and its constructor[2] defined. A member variable $x^{<3>}$ is…

Output from running the code is annotated in a similar way with line numbers and a leading tab sequence of dashes to reference lines of output in the narrative text.

Special Note:
Lines of code are numbered starting with 0 (as opposed to 1) as a reminder indexing in C++ starts with 0!

Text that requires typing at the console is represented as

win>you type this
lin>command line input for linux

where the ">" represents the prompt preceded by an identifier that designates the operating system.

2.3.1.2   Setting Up a Compiler

It is assumed you are familiar with and have a working C++ compiler to proceed. Along with
the code, the commands to work the example with the Borland 5.5 (Windows) and g++ (Linux)
compilers are shown. The examples should compile and run just as easily with any other
standard C++ compiler.

Special Note:
As of this writing, the Borland 5.5 compiler is freely available and can be downloaded at
www.borland.com.

Before proceeding, make sure you can write, compile, and execute a "Hello World" type
program something along the lines of

```
0 --------#include <iostream>;
1 --------using namespace std;
2 --------int main() {
3 --------  cout << "hello CMD" << endl;
4 --------}
```

Create a new directory in the same directory that has the kernel directory (a directory named
"osk") in it. Either the Windows or Linux directories in the distribution are a good place.
Within this new directory, begin with a blank file called hello.cpp.

2.3.2 Coding the Models

Since the simulation will consist of three distinct entities, it is logical to make each entity a C++
class. We'll write individual C++ classes to represent each model. The classes can then be used
to instantiate working objects for the simulation. Construction of the models consists of writing
specific methods that override those in the CMD kernel. Inheriting from and overriding classes
in the kernel provide the facility for the executive in the kernel to call each method at the
appropriate time. The kernel also provides many interfaces to control its operation with respect
to the models.

2.3.2.1   Target

The target model is the simplest so we'll start with it.

2.3.2.1.1   Building

First, the *Target* class[0] is declared:

```
0 --------class Target : public Block {
1 -------  public:
2 --------    Target( double x, double y, double vx, double vy) {
3 --------      this->x = x;
4 --------      this->y = y;
5 --------      this->vx = vx;
6 --------      this->vy = vy;
7 --------      addIntegrator( this->x, this->vx);
8 --------      addIntegrator( this->y, this->vy);
9 --------    }
10-------    double x, y, vx, vy;
11-------    void rpt() {
12-------     if( State::sample( 1.0) || State::tickfirst || State::ticklast) {
13-------       printf( "%12s %8.3f %8.3f %8.3f\n", "Target", State::t, x, y);
14-------     }
15-------    }
16-------};
```

The class is derived from the *Block* class[0] which provides the model with the functionality of the kernel. The kernel takes care of a lot of complex actions as the simulation executes – but all this is hidden by simply inheriting functionality from the *Block* class.

Since the target is constant velocity in two channels, four variables are needed to define the velocity and position, and are declared as the class instance variables *x*, *y*, *vx*, and *vy*[10]. The initial values of these parameters are passed in the constructor[2]. Within the constructor, the variables are set to their initial values[3-6]. The keyword *this*[3-6] is used to distinguish the class instance variables from those passed in the constructor's calling argument list.

Two integrators need to be added since the target model has two states (position in x and y channels). This is easily accomplished with the kernel's *addIntegrator*[7,8] method. The first argument is the state and the second its derivative. This is all that needs to be done for integrating (i.e. propagating) the states – the kernel takes care of the rest!

Special Note:
The kernel internally creates (instantiates) a *State* object when *addIntegrator* is called.

Console output as the simulation executes is useful for observing the operation of the target model. The kernel calls the *rpt* method[11] periodically at the end of each full integration time-step as the states are propagated. One option would be to simply put a *printf* statement directly inside the *rpt* method. The downside to this would be that output would then occur at every integrating time-step (including sub-steps for multiple evaluation integration algorithms), leading to excessive output. The kernel provides a very flexible *sample*[12] function that can be used to

specify output at specific intervals and times. *sample* is a boolean function that returns true if two conditions are met: (1) Time is at the beginning of an integrating time-step, and (2) Time is at an even increment of the argument to *sample*. In this case, the boolean will be true (and output occurs) every second. What if a particular model doesn't necessarily begin or end at an even increment of the sample time? The kernel takes care of that too with *tickfirst* and *ticklast* class variables[12]. The class variable *tickfirst* is true if it is the first time the model is called. The class variable *ticklast* is true if the simulation as a whole is terminated for any reason. This kernel variable is useful for capturing that last simulation output, even if it occurs outside of a regular print interval.

Special Note:

Notice the distinction between *instance* variables and *class* variables. A class serves as the definition for creating objects. Objects are created or *instantiated* from the class definitions. Multiple, independent objects can be created from a single class definition where each object has its own unique set of *instance* variables (like *x*, *y*, *vx*, *vy* in the *Target* class). A *class* variable is a variable associated with a class, not an instance of a class. A class variable is owned by all instances of a class; if one object changes the variable, it is changed in all the objects. *State::tickfirst* and *State::ticklast* are *class* variables to give the kernel the capability to change these values for all the state objects.

As can be seen from this example, writing a model consists of building a standard C++ class that inherits functionality from the kernel by inheriting *Block*. Functionality is obtained by overriding methods that are called by the executive within the kernel and using kernel variables and functions to specify how the model is to operate.

2.3.2.1.2   Testing

If you haven't already typed in the code for the Target class, complete doing so.

To verify everything is typed in correctly, let's test the target model by compiling it. To do this, we need to build a main<0> program:

```
0 --------int main() {
1 --------   Target *target = new Target( 20.0, 5.0, -1.0, 0.0);
2 --------}
```

Then create a new target object<1> from the class we have just created. The object is created using a pointer and the C++ dynamic allocator new function. In general, all objects should be created as pointers so as they get passed around, only the address of the object (and not all the contents of the object) needs to be passed.

The kernel API prototypes need to be declared so make this the first statement in the program:

```
0        #include "../osk/sim.h"
```

Make sure the osk directory is in the parent directory (this will be the case if you put hello.cpp in any of the distribution's examples directory).

The code is compiled by typing the compilation commands at the console command line:

dos>bcc32 hello.cpp ../osk/state.cpp ../osk/block.cpp ../osk/sim.cpp
lin>g++ hello.cpp ../osk/state.cpp ../osk/block.cpp ../osk/sim.cpp -o hello

You may get a warning message that the target object has been created but not used – that is no problem if there are no other error messages.

Special Note:
The compilation process can be greatly automated over the above method of simply typing the commands at the command line. Many editors have macro facilities for making short-cuts to frequently used commands. Makefiles are also a good way of expediting the compilation and linking process.

The compilation command gives a distinct clue on how the kernel is put together. Classes are defined at all levels: the *State* class (in state.cpp) for the integrator states, the *Block* class (in block.cpp) for the models, and the *Sim* class (in sim.cpp) for the simulation as a whole. Every bit of the kernel's functionality is encapsulated in a hierarchy of objects.

2.3.2.2   Missile

Construction of the missile model proceeds similarly to that of the target. A class encapsulating all the missile model's state variables and functionality is defined.

2.3.2.2.1   Building

The full missile class is declared as:

```
0 --------class Missile : public Block {
1 -------- public:
2 --------    Missile( Target *target, double x, double y, double vel) {
3 --------      this->x = x;
4 --------      this->y = y;
5 --------      this->vel = vel;
6 --------      this->target = target;
7 --------      addIntegrator( this->x, vx);
8 --------      addIntegrator( this->y, vy);
9 --------    }
10-------   void update() {
11-------      double dx = target->x - x;
12-------      double dy = target->y - y;
13-------      d = sqrt( dx * dx + dy * dy);
14-------      vx = vel * ( target->x - x) / d;
15-------      vy = vel * ( target->y - y) / d;
16-------      if( d <= 0.1) {
17-------        Sim::stop = -1;
18-------      }
19-------   }
20-------   void rpt() {
21-------     if( State::sample( 1.0) || State::tickfirst || State::ticklast) {
22-------       printf( "%12s %8.3f %8.3f %8.3f %8.3f\n", "Missile", State::t, x, y, d);
```

```
23-------    }
24-------    }
25-------    double x, y, vx, vy, vel, d;
26-------    Target *target;
27-------};
```

As always, without exception, the model class derives from the kernel class *Block*[0]. Model variables are passed in the constructor to be initialized[2]. Like the target model, the missile has two state variables (position in each channel) along with their corresponding derivatives [25]. These are used to define two integrators [7,8].

The state derivatives were constant in the target model and hence only needed to be initialized in the constructor because they didn't change as the states were propagated. What if the derivatives need to be updated with respect to time (which they usually are)? The kernel provides an *update* [10] method to be overridden for defining the derivatives. The kernel's default *update* method simply integrates any state derivatives using their values specified in the *addIntegrator* method if *update* is not overridden. Here, the missile's state derivatives are updated [11-15] with respect to time in accordance with the pursuit equations previously shown.

Another requirement which distinguishes the missile from the target model is the requirement to access the target's states. These are needed to define the pursuit equation derivatives in *update* [13-15]. To accomplish this, a pointer reference to a target object is declared as a class instance variable [26]. It is passed in the constructor argument list and set in the constructor [2]. These references are then used to access the target states[11,12,14,15].

Special Note:
You might have noticed that all the parameters in each class are declared as *public*, meaning any external class can have full access to the parameters and methods. In larger scale simulations, it might be desirable to restrict access to some class instance variables (and/or methods). This is normally done by declaring the variables *private* (or *protected*) and then defining *access* functions that return the variables values (so they can't be overwritten). We simply access the variables directly here for simplicity of illustration. The kernel provides a shorthand mechanism for quickly defining and using access functions (which is described later).

Range-to-go is tabulated and polled to terminate the simulation when it becomes less than a threshold [16]. The kernel provides a class variable *Sim::stop* [17] which can be used, among other things, to terminate the simulation. Setting this variable to a value less than zero instructs the kernel to terminate the simulation at the conclusion of the current integrating time-step.

The *rpt* method [20], in conjunction with the *sample* function, is used to report output at a prescribed time increment.

2.3.2.2.2 Testing

If you haven't already typed in the code for the *Missile* class, complete doing so.

Let's compile the class to ensure there are no typographical errors. First, create a *missile* object from the class we've just created in *main* [2].

```
0 --------int main() {
1 -------   Target *target = new Target( 20.0, 5.0, -1.0, 0.0);
2 -------   Missile *missile = new Missile( target, 0.0, 0.0, 2.0);
3 -------}
```

The compilation directive is the same as before.

dos>bcc32 hello.cpp ../osk/state.cpp ../osk/block.cpp ../osk/sim.cpp
lin>g++ hello.cpp ../osk/state.cpp ../osk/block.cpp ../osk/sim.cpp -o hello

Again, the compiler might warn that the *missile* object is not used. That is alright if no errors are reported.

2.3.2.3   Radar

The function of the radar in the hypothetical system is to point at the target. Once the "beam" (pointing direction) of the radar is within 1 degree of the target, the missile should be launched. To illustrate a dynamic system that is slightly more complex than simply integrating higher-order derivatives down to their lower-order states (velocity to position, that is), the beam pointing dynamics are modeled as a second-order system with feedback as described earlier in the problem statement and illustrated in *Figure 4.* The time-of-response is governed by the frequency, ω_n, and the damping, ζ. The values shown in the figure were selected to get a 1 second 90 percent response time to steady-state. The motivation for including this simplified servo dynamics model is to illustrate CMD application for a different domain of differential equation applications.

$$\frac{y}{u} = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

$$90\% \text{ response time}, t_r = \frac{7.04\zeta^2 + 0.25}{2\zeta\omega_n}$$

$$\omega_n = 2.643, \zeta = 0.7 \text{ for } 1 \sec 90\% \text{ response}$$
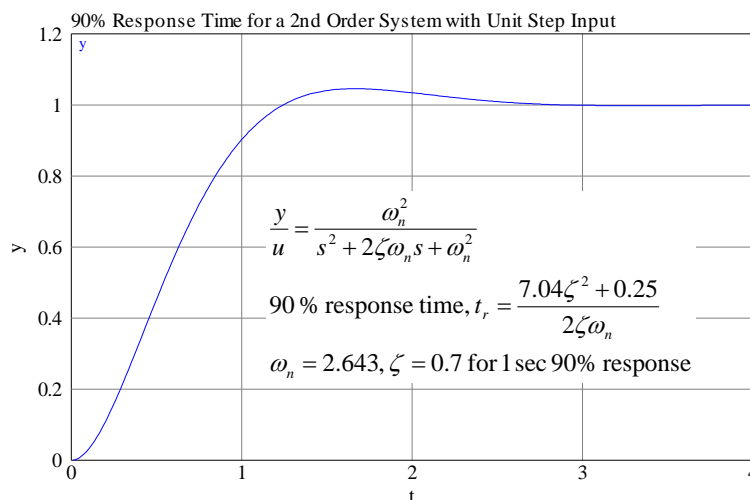
Figure 4. Second-Order System Response to a Step Input

16

2.3.2.3.1 Building

The radar model is encapsulated as a class.

```
0 --------class Radar : public Block {
1 --------  public:
2 --------    Radar( Target *target, double theta, double wn, double zeta) {
3 --------      x1 = theta / R;
4 --------      x2 = 0.0;
5 --------      this->wn = wn;
6 --------      this->zeta = zeta;
7 --------      this->target = target;
8 --------      addIntegrator( x1, x1d);
9 --------      addIntegrator( x2, x2d);
10-------    }
11-------    void update() {
12-------      double theta_target = atan( target->y / target->x);
13-------      theta_err = ( theta_target - x1);
14-------      if( fabs( theta_err * R) < 1.0 && Sim::stop == 0) {
15-------        Sim::stop = 1;
16-------      }
17-------      x1d = x2;
18-------      x2d = theta_err * wn * wn - 2.0 * zeta * wn * x2;
19-------    }
20-------    void rpt() {
21-------      if( State::sample( 1.0) || State::tickfirst || State::ticklast) {
22-------        printf( "%12s %8.3f %8.3f\n", "Radar", State::t, theta_err * R);
23-------        printf( "\n");
24-------      }
25-------    }
26-------    double x1, x1d, x2, x2d, theta_err, wn, zeta;
27-------    Target *target;
28-------};
```

The *radar* class only introduces one new kernel feature [15] so it should be pretty clear what is going on.

To reiterate, the class gets its functionality from the kernel by deriving from *Block* [0]. Variables for initialization are passed in the constructor [2-7]. A global variable $R = \dfrac{180}{\pi}$, that we will add later, aids conversion between radians and degrees. Two new integrators [8,9] are defined for the states *x1* and *x2*. The state equations for the servo [17,18] are defined in the *update* method. The pointing error of the radar [13] is defined as the difference between where the radar is pointing, *x1*, and the angle to the target [12]. Finally, the *rpt* method [20-24] takes care of output.

The remaining issue, and a big one at that, is how do we signal the missile to launch? Remember, the missile should be launched when the pointing error becomes less than a threshold (1 degree, in this case). Resolving this issue is where the CMD architecture really shows its power. One way to address this problem is to have all three models executing concurrently (target, missile, and radar) with the missile simply idling until it receives its "launch" signal, at which time it starts propagating its state equations. The problem with this approach is that the logic to accomplish this must be built into the missile model (class) itself. Built-in logic for this

unique function starts to specialize the missile model for this particular application, potentially reducing the model's reuse elsewhere. Now, everywhere else the model is used, this idle capability must be addressed, whether it is used or not.

The CMD kernel's architecture provides a quite different, albeit elegant, solution to this problem. The simulation will run in two sequential stages. During the first stage, only the target model and radar model will execute. Upon receiving a signal to activate the missile from the radar model, the simulation will enter the second stage of execution where the missile model will be executing in addition to the target and radar. The real power to this approach is that simulation model selection and control logic is *external* to the models (and, in this case, is automatically supplied by the kernel). The only thing that has to be included in the models themselves is the event triggering logic telling the kernel to switch the models' execution. Let's defer exactly how to set this up in the kernel to the next section. For now, all we need to know is how to signal the kernel to change the model execution sequence. This is another application of the kernel's *Sim::stop* class variable. Here, instead of using the variable to stop the simulation (like we did in the missile model), we use it to tell the kernel to change the model execution sequence. This is done by changing the current value of *Sim::stop*[15] from its current value (which is initialized to 0 by the kernel when the simulation starts).

Special Note:
Care needs to be taken when signaling discrete events by changing *Sim::stop*. The variable *theta_err* will be less than the threshold on many subsequent calls to *update* after the threshold is first met. To avoid "clobbering" *Sim::stop* assignments that may occur in the other models (*Sim::stop* = -1 to end the simulation for instance, in the missile model), steps should be taken to make the assignment of *Sim::stop* occur only once. This is easily accomplished by adding the second condition in the *if* statement.

Again, let's defer exactly how the model execution sequence is setup in the kernel until we talk about the main program.

2.3.2.3.2 Testing

If you haven't already typed in the code for the *Radar* class, complete doing so.

While we are not actually going to run the radar model yet, let's make sure it will compile successfully. First, we need to add a definition for our radians-to-degree conversion factor *R*.

```
0 --------const double PI = 4.0 * atan( 1.0);
1 --------const double R = 180.0 / PI; // rad to deg conversion factor
```

Put this right before the *Radar* class definition.

Then create a *radar* object[5] from the class we've just created in *main*.

```
2 --------int main() {
3 --------   Target *target = new Target( 20.0, 5.0, -1.0, 0.0);
4 --------   Missile *missile = new Missile( target, 0.0, 0.0, 2.0);
5 --------   Radar *radar = new Radar( target, 60.0, 2.643, 0.7);
6 --------}
```

Make sure hello.cpp still successfully compiles.

dos>bcc32 hello.cpp ../osk/state.cpp ../osk/block.cpp ../osk/sim.cpp
lin>g++ hello.cpp ../osk/state.cpp ../osk/block.cpp ../osk/sim.cpp -o hello

2.4 Building the Main Program

Up until now, we have focused on building the individual model classes. It is in the main program that we assemble the simulation, again using API from the kernel. Here, we will fully describe, assemble, and execute the models.

The main program, bringing everything together, is quite short and concise:

```
0 --------int main() {
1 --------   double  tmax = 10.00;
2 --------   double  dt = 0.01;

3 --------   Target *target = new Target( 20.0, 5.0, -1.0, 0.0);
4 --------   Missile *missile = new Missile( target, 0.0, 0.0, 2.0);
5 --------   Radar *radar = new Radar( target, 60.0, 2.643, 0.7);

6 --------   vector<Block*> vObj0;
7 --------   vObj0.push_back( target);
8 --------   vObj0.push_back( radar);

9 --------   vector<Block*> vObj1;
10-------   vObj1.push_back( target);
11-------   vObj1.push_back( missile);
12-------   vObj1.push_back( radar);

13-------   vector< vector<Block*> > vStage;
14-------   vStage.push_back( vObj0);
15-------   vStage.push_back( vObj1);

16-------   double dts[] = { dt, dt};
17-------   Sim *sim = new Sim( dts, tmax, vStage);
18-------   sim->run();
19-------}
```

The declarations for creating the new objects [3-5] have already been completed when we compiled each model.

Let's look at what is going here from a "macro" perspective. First, a vector (as the name implies) is a container that holds objects. These objects can be of any kind as long as they are derived from the kernel's *Block* class. We have created two vectors of objects: *vObj0* [6] and *vObj1* [9]. The objects will be initialized, propagated, and report out in the order that they are added to the vector (first-in, first-to-execute). Now note that we define a third vector *vStage* [13]. We add to this vector the two vectors of objects we have already created, *vObj0* [14] and *vObj1* [15].

> Special Note:
> The vector container is provided by the C++ Standard Template Library (STL) and not the kernel.

Can you guess how this hierarchical composition of objects translates into execution sequence of the objects? Within the vector *vStage*, the sequence of objects in *vObj0* are first executed in the order that they were loaded in *vObj0*. That is, they are first initialized and then their *update* methods are called in the integrating time loop, along with their *rpt* methods. Upon getting a signal from the kernel (via the *Sim::stop* kernel class variable), the sequence of objects in *vObj1* are then executed. Conceptually, it is convenient to think of the vectors of objects in *vStage* as stages, as in stages of the simulation or stages on a rocket, although each stage can represent anything (not necessarily stages on a rocket).

It should be apparent now how we achieve our objective of first executing only the target and radar models and then, with a sudden change in the topology of models in the simulation, execute all three models. This simple hierarchical train-of-objects paradigm is remarkably powerful and flexible. Very complex simulations can be readily mapped into this representation as well as a very simple simulation consisting of only one object – one object in a stage and one stage in the simulation. The leap from a single object simulation to simulations with complex sets of objects (many objects in a stage and many stages) is remarkably seamless and transparent in this architecture. The organization of the code is apparent and elegant enough that no graphical diagram is really necessary to depict the organization (the code is self-documenting).

Now let's explain some details. The kernel provides a *Sim* class to execute the stages in *vStage* (or whatever we choose to name it). We pass it a hierarchical collection of objects or "train-of-objects" to execute in the constructor [17]. We also have to tell the kernel what integrating time-step to use as well as an absolute ending time (if one of the models doesn't stop the simulation first). Distinct values of *dt* can be used for each stage. Having created a *sim* object, the kernel provides a run method to execute the simulation. Yes, the whole simulation itself is an object! Remember the radar model changing the class variable *Sim::stop* when the target was within a pointing error threshold? This was simply an event trigger to the kernel to transition to the next stage (i.e. a signal to start executing the models in *vObj1*). Since the target and radar models in *vObj0* and *vObj1* are one-and-the-same, they continue to execute along with initiation of the missile model.

> Special Note:
> The integrating time increments within a stage are not restricted to those passed to the *Sim* class. Advanced usage of the kernel's *sample* function can be used to vary the integrating time increment within a stage, even including the capability for *asynchronous* sampling. This is discussed later in one of the tutorial sections.

2.5 Running the Program

If you haven't already typed in the code for the main program, complete doing so.

The program is compiled as before.

dos>bcc32 hello.cpp ../osk/state.cpp ../osk/block.cpp ../osk/sim.cpp
lin>g++ hello.cpp ../osk/state.cpp ../osk/block.cpp ../osk/sim.cpp -o hello

---

Special Note:
All the code in this example was built in one single source code file for simplicity of illustration.
Normally, the objects are put in individual files with an associated header file containing the
class declaration prototype. Instructions for compilation and linking are declared in a makefile.
The other examples included in the distribution, and used later in this manual, are constructed
using the more commonly applied header file and body file organization.

Commands to run the program are

dos> hello
lin> hello

The program executes and the *rpt* methods within the *target*, *radar*, and *missile* objects are
executed. Output from each object occurs in the order that the object was put into each stage.
The resulting output is shown. The first output from each object has been annotated with the
variables in the *printf* statement for convenience in referencing the variables in the printout.

```
0 --------    Target   0.000  20.000   5.000                 "Target", State::t, x, y
1 --------     Radar   0.000  -45.964                        "Radar", State::t, theta_err * R

2 --------    Target   1.000  19.000   5.000
3 --------     Radar   1.000   -4.081

4 --------    Target   1.150  18.850   5.000
5 --------   Missile   1.150   0.000   0.000   19.502        "Missile", State::t, x, y, d
6 --------     Radar   1.150   -0.927

7 --------    Target   2.000  18.000   5.000
8 --------   Missile   2.000   1.641   0.446   16.982
9 --------     Radar   2.000   2.005

…more output…

10-------    Target   6.000  14.000   5.000
11-------   Missile   6.000   9.228   2.965   5.188
12-------     Radar   6.000   0.665

13-------    Target   7.000  13.000   5.000
14-------   Missile   7.000  11.023   3.843   2.290
15-------     Radar   7.000   0.755

16-------    Target   7.810  12.190   5.000
17-------   Missile   7.810  12.189   4.913   0.087
18-------     Radar   7.810   0.8420
```

Let's briefly examine the output to ensure the models operated as anticipated. First note that output for each object appeared periodically at the specified time increment of 1.0 second (using the kernel's *sample* function), with the exception of the "staging" event at 1.150 seconds and the termination time at 7.810 seconds. The radar was initially pointing at 60 degrees (as initialized in its constructor). The target was initialized (in its constructor) at a range of 20 and altitude of $5^{<0>}$ (or 14.036 degrees off of the horizon). From this, the initial pointing error of the radar is 14.036 degrees – 60 degrees = 45.964 degrees as reported[1]. The target gets closer[2] and the radar's pointing error is reduced by the servo as the simulation progresses[3]. The radar's pointing error is reduced to less than the threshold of 1 degree[6] and triggers the missile to be launched[5] at 1.150 seconds. What is actually happening in the simulation is that the train-of-objects being executed by the kernel progresses from the first stage (*vObj0*) to the second stage (*vObj1*) triggered by changing the value of *Sim::stop* in the radar model. The target and radar objects continue to propagate from their previous states in *vObj1* and the missile model is started. The missile flies a pure pursuit trajectory and gets closer and closer to the target as evidenced by the range-to-target, *d*, decreasing[8,11,14]. When the range-to-target becomes less than $0.1^{<17>}$, the simulation terminates in response to *Sim::stop* being set to -1 in the missile model. Note that the missile launch printout and the final time printout were obtained using the kernel's *State::tickfirst* and *State::ticklast* parameters, respectively.

2.6 Going Further

This example fully illustrates the operational aspects of CMD and its kernel. The kernel provides a lot more utility, but the concepts illustrated in this example show, from a top-level perspective, the full scope of the kernel's application. It is remarkable that such a flexible operational capability can be captured in such a concise coding architecture.

Within the train-of-objects executed by the kernel, straightforward mechanisms are provided to fine-tune the execution of the objects between stages. If the same object is used between stages, simple and flexible kernel commands are provided to continue execution of the object starting at its final state in the previous stage or reinitializing it. An object's state can even be temporarily frozen by including it in a stage, leaving it out in the next stage, and then putting it back in the next stage. The option of customizing an object's initialization at the beginning of every stage provides a very straightforward means of implementing complex object execution sequences and interactions within a relatively simple paradigm.

Having worked through a comprehensive example, the next section takes a step back from actual code and talks about the kernel's architecture to reinforce, at an abstract level, the principles illustrated in this section. Following that, a set of comprehensive examples that successively build on each other illustrate, with code, the kernel's features and how to use them.

3   THE BIG PICTURE

This section describes the generalized architecture behind the concepts illustrated with the example system of the previous section. The objective of this section is not to describe the architecture's design in detail (as in a software design specification), but instead to focus on its organization and operation so that it can be effectively used to build simulations.

The explanations in the previous example should become clearer with the descriptions in this section – it might be helpful to go back and review the previous example after reading this section.

3.1 The OSK

The heart of CMD is the powerful OSK. A kernel is defined as an "essential subset of a core function in terms of which other constructs for accomplishing a function can be defined." The CMD kernel provides the core functionality of solving the Differential Equations (DEs) and provides the constructs for building models to represent the DEs. The kernel performs all simulation infrastructure functions including model initialization, updating, state propagation, model sequencing, execution control, and reporting of output.

The kernel consists of a state class to represent the states within the DEs, an abstract base class for constructing the models, and an executive to execute the models and their associated methods in the right order at the right time. These are described in the following subsections. The following sections then describe the functionality required for each model and how the models are collected to build a simulation.

3.1.1 State

The fundamental entity in a time-based simulation is an integrator. In fact, a simulation can be conceptually viewed as a collection of integrators bound together by algorithms that define their derivatives.

The kernel provides a State class that encapsulates the numerical integration algorithm and associated data to propagate a state forward in time. For simplicity, the user doesn't interact directly with the *State* class. An Application Programming Interface (API), *addIntegrator*, is provided to declare a state (and a variable for its derivative) and have it automatically propagated by the kernel.

Special Note:
A powerful feature of the kernel's design is the ability to add your own numerical integration algorithms using C++'s inheritance mechanism. This advanced feature is described in the Appendix along with instructions for adding your own numerical integration algorithm-of-choice.

3.1.2 Block

The kernel has a Block class that fulfills two purposes: (1) it encapsulates all the data and functionality that defines a model in one program unit, and (2) provides the "hooks" into the models' methods so that they can be controlled by the kernel's executive.

One of the foundations of an O-O representation is that an object encapsulates its methods and data. Thus, it seems logical that, in addition to each model having its own methods, the models have data structures that contain their information. This fits perfectly into the O-O paradigm by defining each model as a class with its own methods and data encapsulated together into a single code unit. Thus each model is defined as a class, and the class serves as a "cookie-cutter" to create (instantiate) objects with specific data.

The kernel's Block class is an abstract base class with "stub" methods for the models' initialization, definition of derivatives, and output reporting. The first step in constructing a model is to derive from this class. This ensures that the model has the interfaces necessary to interact with and be controlled by the kernel.

3.1.3 Simulation

The kernel's simulation class is the simulation executive. It orchestrates, at a top level, calling the models in the desired sequence and, at a lower level, taking care of calling the models' methods (inherited and overridden from the *Block* class) to initialize and propagate the states within each model. The executive also propagates the states forward in time using a numerical integration algorithm.

3.2 Model Method Calling Sequence

A survey of successful current and past simulations of the time-marching DE solving-variety [8] revealed that, regardless of the computer language that they were written in or the specific application, they all had a common thread of functionality. First, the DEs were modularized (to different extents) in code units organized according to functionality, which we've been generally calling models here. Then, each of the models had a common functionality: first they were initialized, then the state derivatives defined, the states propagated forward, and then there was a provision for reporting the current variable states. This takes place over-and-over in a cycle comprising an integrating time loop. The model methods in the integrating time loop are called in an order tailored to the problem being solved so that the state derivatives are defined with the most current values.

As a specific example, math modelers generally represent digital simulations in the form of a block diagram of sequentially executed models. Each model or block corresponds to a subsystem or process that is physically separated from the others by a set of well-defined interfaces. For instance, a homing missile simulation can be represented by the general block diagram in *Figure 5*. Contained in each block are mathematical expressions representing the functionality of that element. For instance, the equations-of-motion block might contain the quaternion derivatives that describe the attitude of the missile. The autopilot block might contain the algebraic expressions relating a commanded acceleration to commanded fin deflections.
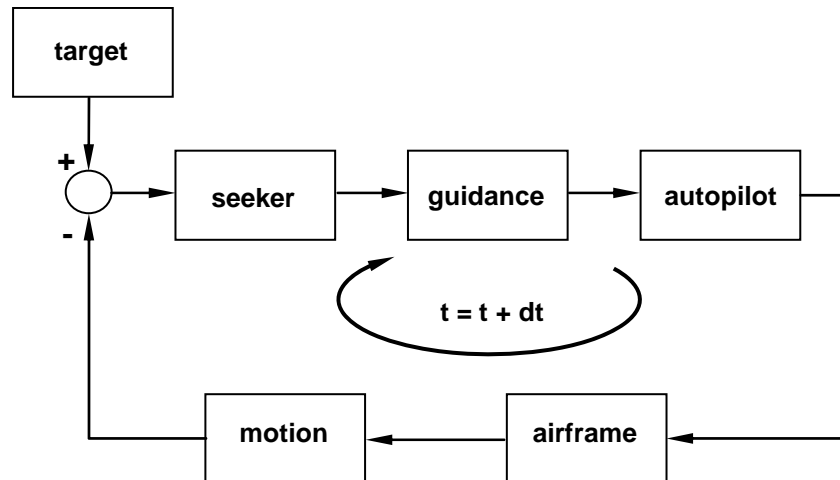
Figure 5. Homing Missile Block Diagram for Simulation

Ideally, the loop in ***Figure 5*** is traversed continuously. In digital simulation, the loop is traversed once every integrating time-step. During each time-step, the derivatives must be defined for all the blocks and then their states propagated to the next time-step. This cycle is repeated until some condition terminates the simulation such as reaching a termination time. Before the loop is traversed at all, each of the states must be initialized.

This notion of operation lends itself nicely and seamlessly to a C++ O-O context, and serves as the basis for defining the common functions that each model object must perform. Each model (its data and functionality) are encapsulated into a C++ class. Within each model, functionality for initialization, updating the derivative definitions, and a provision for output reporting must be provided.

An O-O representation for this flow of control is shown in ***Figure 6***. Each model is encapsulated in an object that has its own unique functionality for each of the common methods previously described. The kernel calls the methods in each model at the appropriate time, in addition to calling its own internal methods to actually perform the numerical integration and other simulation infrastructure functions. This is all mechanized in code by the kernel providing a base class from which to derive the model classes. The kernel knows when to call the appropriate method in the base class so these methods are simply overridden in the derived model classes.
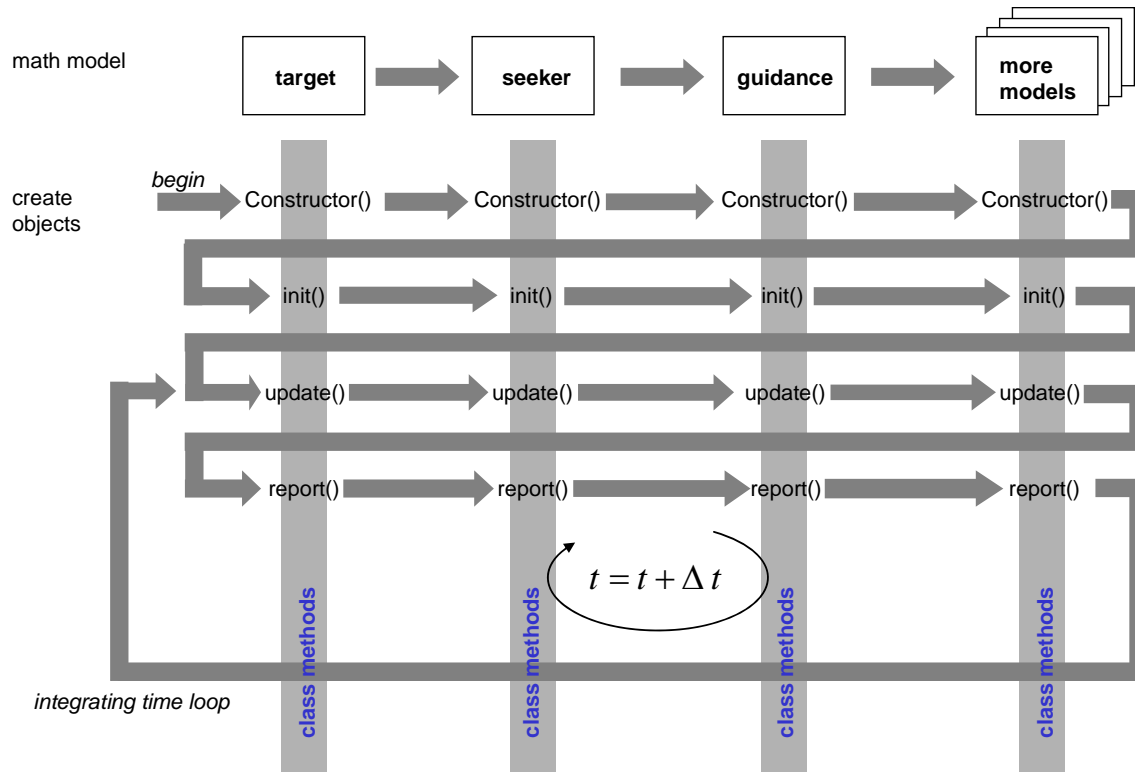
Figure 6. Object-Oriented Model Flow-of-Control

3.3 Train-of-Objects System Architecture

Having defined an O-O strategy for solving the DEs within the models, how is the execution sequence organized? The strategy should not only take into account executing only one particular series of models, but also have the capability to execute multiple sequences of models, to account for event-driven changes in the system's topology (changing stages on a missile for instance). A unique "train-of-objects" approach has been developed to accomplish this [9].

The organization of the entities used by the kernel is shown in *Figure 7*. At a micro-level, each state (that is each individual DE) is a C++ object (STATE in the figure). One or more states are combined, connected by algorithms for their derivatives, into a block (or model). Several models are combined into a stage. Finally, a series of stages are combined in a list. A simulation class in the kernel then decomposes this "list-of-lists-of-lists" to execute the models in the desired order (specified by their order in the lists) and, within each model, the appropriate methods described in the previous section.

INTEGRATOR OBJECT

STATE

BLOCK

MODEL OBJECT

STAGE OBJECT

STAGE

STAGE OBJECT

SIMULATION
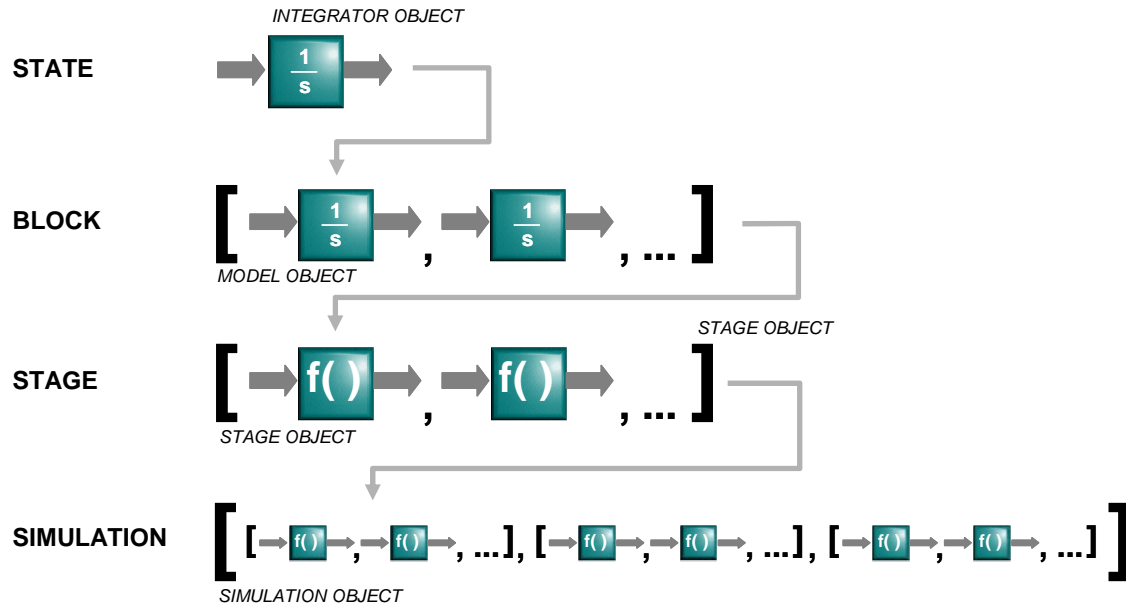
SIMULATION OBJECT

Figure 7. Train-of-Objects Architecture

Thus, the kernel implements the O-O paradigm at all levels: dynamic states are objects, models themselves are objects, stages are objects, and the aggregate simulation is an object.

Special Note:
The "train-of-objects" is actually an artifact of digital simulation. Of course, in the "real" world all these processes are occurring simultaneously. The models and the internal processes they model must be serialized to some degree (even on multiple processor systems) since all computations take place in time-steps.

4   TUTORIAL EXAMPLES

This section fully illustrates, by example, the mechanics of using CMD's many capabilities. The approach is to start with a simple dynamic system and incrementally add capabilities to the model to illustrate CMD feature syntax and usage. We will end up with a working example that demonstrates all the features described in Section 1.4. The feature set illustrated here should encompass all the capabilities you need to build your own simulation.

All the examples are included in the distribution. For convenience, *Table 1* cross-references the examples to their directories in the distribution.

Table 1. Example Location in Distribution

Name	Section	Directory
Simple Airframe/Autopilot Dynamics Model	4.1	ex_1
Mixing Discrete and Continuous Models	4.2	ex_2
Scheduling Events	4.3	ex_3
Building and Connecting Multiple Models	4.4	ex_4
Using Multiple Stages (Changing Model Topology)	4.5	ex_5
Capturing Asynchronous Events	4.6.3.1	ex_6a
Using sample to Generate Time-Steps	4.6.3.2	ex_6b
Using sample to Control the Time-Step	4.6.3.3	ex_6c
Comprehensive Sample Demonstration	4.6.3.4	ex_6d

The examples are ready to be compiled and executed in the distribution. Ready-to-use makefiles for both Windows (using the Borland 5.5 compiler) and Linux (using g++) are also included. Simply type *make* at the command line to compile.

dos>make
lin>make

For brevity and ease-of-presentation, only the pertinent code that is being described is included in the discussion. The full code listings can be consulted in the distribution. Probably the best way to understand the features each example illustrates is to first consult the code listing fragment referred to in the narrative, and then study the full code. After this, running each example and experimenting with the code is highly encouraged.

4.1 Simple Airframe/Autopilot Dynamics Model

4.1.1 Building the Model

The purpose of this starting example is to simply get a model in-place to serve as the basis for adding and illustrating features in subsequent sections so we will quickly go through this. We start off with a simple airframe steering example as shown in *Figure 8*.
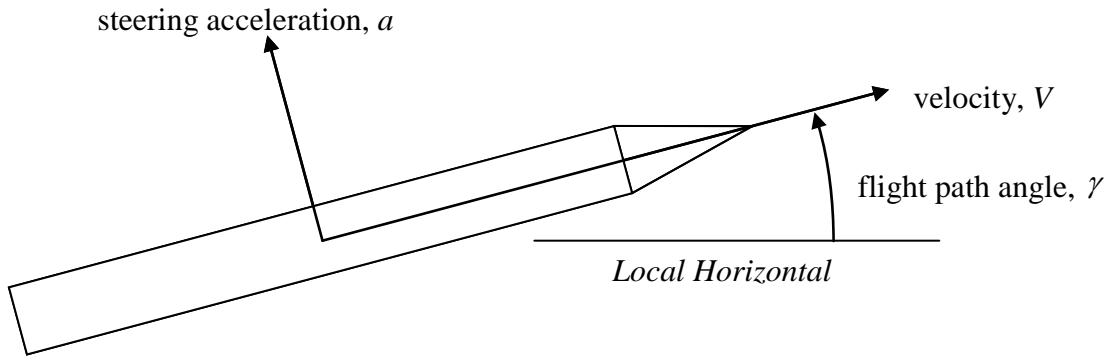
Figure 8. Simple Missile Steering Dynamics Model

The missile is flying at constant velocity V. The objective is to adjust the steering acceleration, *a*, to steer the missile along the desired flight path angle, γ. The equation of motion for the flight of the missile is

$$\dot{\gamma} = \frac{d\gamma}{dt} = \frac{a}{V}$$

A feedback controller can be "wrapped" around the airframe as shown in the block diagram in *Figure 9*.
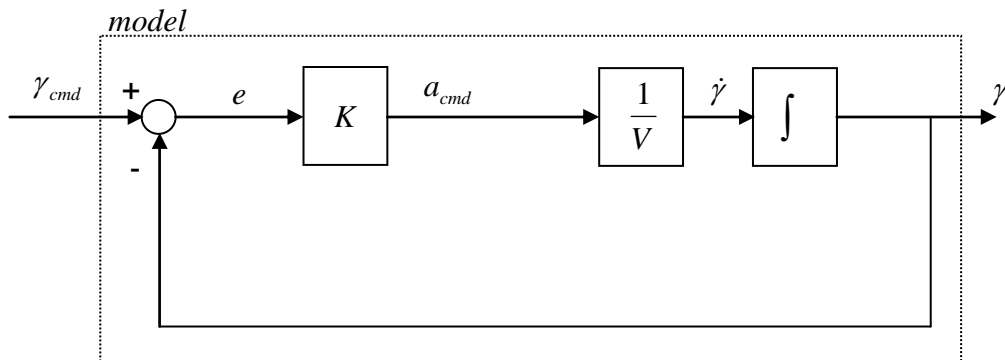
Figure 9. Missile Dynamics Steering Model with Closed-Loop Control

The equations for the closed-loop system are easily derived directly from the block diagram

$$e = \gamma_{cmd} - \gamma$$

$$a_{cmd} = Ke$$

$$\dot{\gamma} = \frac{a_{cmd}}{V}$$

$$\gamma = \int \dot{\gamma}\, dt$$

The closed-loop transfer function is

$$\frac{\gamma}{\gamma_{cmd}} = \frac{\dfrac{K}{Vs}}{1 + \dfrac{K}{Vs}} = \frac{1}{\dfrac{Vs}{K} + 1} = \frac{1}{\tau s + 1}, \tau = \frac{V}{K}$$

Where τ is the time constant of a first-order system representing the time it takes the response, γ, to reach 63.2 percent of its steady-state value to a step input, γ_{cmd}.

Construction of a CMD model for this process begins with the main program, main.cpp.

```
0 --------int main() {
1 --------   double  tmax = 2.00;
2 --------   double  dt = 0.01;

3 --------   Model *model = new Model( 0.0);

4 --------   vector<Block*> vObj0;
5 --------   vObj0.push_back( model);

6 --------   vector< vector<Block*> > vStage;
7 --------   vStage.push_back( vObj0);

8 --------   double dts[] = { dt};
9 --------   Sim *sim = new Sim( dts, tmax, vStage);
10-------   sim->run();
11-------}
```

The system is relatively simple so we will start off by coding everything in one model (this will be decomposed into multiple models to illustrate model communication later). Everything here should be clear after having worked through the "Hello World" example. The train-of-objects structure must still be used, even though there is only one model – one model in the stage and one stage in the simulation.

> Special Note:
> Even though it adds slightly more coding for very simple systems, the train-of-objects approach assures that this simple simulation will easily scale to more complexity in the future, if required. In any case, the added coding burden is minimal.

Let's look at the model code itself in the file model.cpp.

```
0 --------Model::Model( double gamma_) {
1 -------  gamma0 = gamma_;
2 -------  addIntegrator( gamma, gammad);
3 --------}

4 --------void Model::init() {
5 -------  cout << "starting Model...\n";
6 -------  if( initCount == 0) {
7 -------    gamma = gamma0;
8 -------     k = 1000.0;
9 -------  }
10-------}

11-------void Model::update() {
12------- gamma_cmd = 1.0;
13------- v = 1000.0;
14------- a_cmd = k * ( gamma_cmd - gamma);
15------- gammad = a_cmd / v;
16-------}

17-------void Model::rpt() {
18------- if( State::sample( 0.1)) {
19-------   printf( "%8.3f %8.6f\n", State::t, this->gamma);
20-------  }
21-------}
```

Again, everything should be familiar except the addition of the *initCount* feature[6]. This is not really needed here, but is included as a step toward showing new features. The parameter *initCount* is a model instance variable obtained by deriving from *Block* when the model was created. Each model has its own independent value of *initCount* that is set by the kernel. The *initCount* gets incremented by 1 every time the model's *init* method is called and completed. Providing a counter every time the model is used in a stage gives the model a means to control its initialization strategy based on where it is in the train-of-objects. We will talk a lot more about kernel facilities for staging later with an actual example.

4.1.2 Running the Model

The gain, *k*, was selected for a time constant of 1 second. For convenience, makefiles are provided for all the examples (in their respective directories) so compiling and linking the example is accomplished

dos>make
lin>make

and running it

dos>main
lin>main

The results are printed to the console

```
22-------starting Model...
23-------   0.000 0.000000
23-------   0.100 0.095163
24-------   0.200 0.181269
25-------   0.300 0.259182
26-------   0.400 0.329680
27-------   0.500 0.393469
28-------   0.600 0.451188
29-------   0.700 0.503415
30-------   0.800 0.550671
31-------   0.900 0.593430
32-------   1.000 0.632121
33-------   1.100 0.667129
34-------   1.200 0.698806
35-------   1.300 0.727468
36-------   1.400 0.753403
37-------   1.500 0.776870
38-------   1.600 0.798103
39-------   1.700 0.817316
40-------   1.800 0.834701
41-------   1.900 0.850431
42-------   2.000 0.864665
43-------   2.000 0.864665
```

Output occurred every 0.1 second as specified with the *sample* statement in model.cpp[18]. As expected, *gamma* reached 63.2 percent of the commanded step input at t = 1 second[32], the system's time-constant.

Special Note:
Output occurs twice at the end of the simulation [42,43] if the sample time is an even increment of the *tmax* variable that was passed to the *Sim* class constructor. This signifies that the simulation successfully terminated at this sample time. If *tmax* is not an even increment of the sample time used in the *rpt* method, a special *ticklast Sim* class variable needs to be used to poll for output. This is discussed in a subsequent example.

4.2 Mixing Discrete and Continuous Models

Perhaps the most fundamental way to categorize simulation model operation is that of being either *continuous* or *discrete*. Continuous models evolve in time according to their state derivatives. Discrete models change state at certain points in time (periodically or aperiodically). The discrete models states can evolve in time (maybe as a function of the previous states) or can be set by events. Discrete models are sometimes referred to as *sampled data models*. Models can be entirely continuous or discrete, or a mixture of these can exist within one model (a *hybrid* model).

It is a fundamental simulation requirement to be able to accommodate both these kinds of models. This section illustrates adding a discrete sampler to the missile/autopilot model.

4.2.1 Full-up Model

Let's modify the "autopilot" in the previous example to make it discrete (sampled). This new model is shown in *Figure 10*.
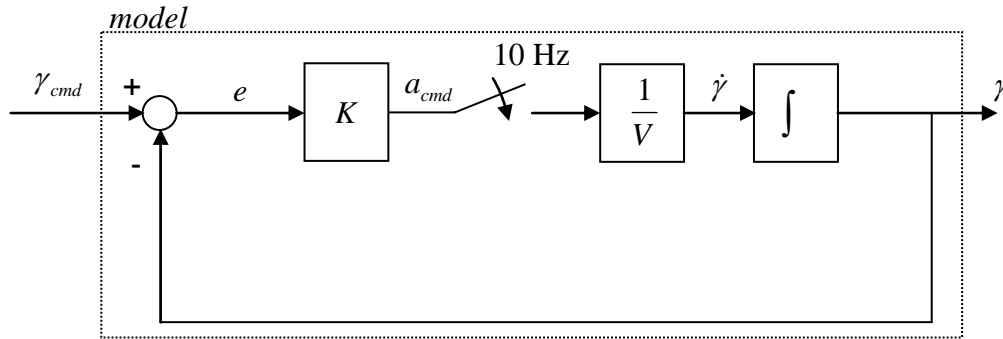
Figure 10. Mixed Continuous and Discrete Model

The model is identical as before except a 10 Hz sampler has been added for the lateral acceleration command. The command is a sample-and-hold operation so that a new acceleration command gets issued every 0.1 second.

CMD has a simple but flexible solution for addressing this very common modeling situation: A special *sample* API is provided that allows seamless mixing of discrete and continuous operations. This is the one-and-the-same *sample* function that heretofore has been used to poll for specific output times in the *rpt* method.

The *sample* function can be used within the *update* method to define periodic discrete samples in identically the same way it is used within the *rpt* method. The *sample* function is a boolean function with the prototype declaration

static int sample(double dt, double t_event = 0);

It returns a 0 (false) or 1 (true). True is returned if it is the beginning of the integrating time-step and the time is an even increment of *dt.*

Special Note:
We will defer discussion of the usage of the second argument *t_event* (which is set to a default value) to a following example. This argument adds a very powerful capability to schedule events "on-the-fly." The second argument can be ignored in this example and, of course, need not be included in the calling statement since its default value is specified.).

Only the *update* method for the *Model* class needs to be changed to reflect this new capability. The code fragment from model.cpp is

```
0 --------void Model::update() {
1 --------  gamma_cmd = 1.0;
2 --------  v = 1000.0;
3 --------  if( State::sample( 0.1)) {
4 --------    a_cmd = k * ( gamma_cmd - gamma);
5 --------    cout << " sample " << State::t << " " << a_cmd << endl;
6 --------  }
7 --------  gammad = a_cmd / v;
8 --------}
```

The acceleration command computation is simply enclosed in an *if* block using the *sample* function[3-6]. A new value of a_cmd[4] is calculated every 0.1 second. A print statement (*cout*) is included so that we may observe the operation of the discrete sampler[5]. Again, the *sample* function makes it easy to precisely place the data sampling code in relation to the continuous calculations (*before gammad* is defined[7]).

Special Note:
You may have noticed that the syntax for *sample* function usage is "*State::sample*" and that the time variable *t* is "*State::t*". These are a *class* method and *class* variable, respectively, in the OSK. Class methods and variables are the same (i.e. global) for all objects created from the *State* class. For instance, *State::t*, is one single variable that is available to all *State* class objects. If one object changes it, it is changed in *all* the objects.

The model is compiled and linked as before

dos>make
lin>make

and ran

dos>main
lin>main

The resulting output is (some has been omitted for clarity):

```
9 --------starting Model...
10-------  sample 0 1000
11-------    0.000 0.000000
12-------  sample 0.1 900
13-------    0.100 0.100000
 ...
14-------  sample 0.9 387.42
15-------    0.900 0.612580
16-------  sample 1 348.678
17-------    1.000 0.651322
18-------  sample 1.1 313.811
19-------    1.100 0.686189
 ...
20-------  sample 2 121.577
21-------    2.000 0.878423
22-------    2.000 0.878423
```

The sampling of the acceleration command at 10 Hz is clearly seen [10,12,14,16,18,20] (with the *cout* statement we added).

More than one *sample if* block can be used in any model's *update* method. Different *sample if* blocks with different sample rates can be included anywhere within the *update* method to get the functionality required. Numerous sequences of continuous calculations intermingled with discrete calculations are easily programmed in this manner in a single *update* method. Multiple chains of intermixed continuous and discrete operations would be very difficult to implement correctly if they had to be segregated into sequentially executed dedicated discrete and continuous *update* methods.

Of course, dedicated continuous and discrete models are easily constructed. A model is entirely continuous if no *sample* functions are present. Wrapping all the calculations in a *sample* function makes a dedicated discrete, sampled model.

We have addressed the requirement for periodic, sampled operations here. What if we want a *one-time* event? The next example addresses this capability but, first, we elaborate on very critical issue.

4.2.2 The Synchronization Issue

Getting model state variable derivatives and discrete events out of sync with the state variables is perhaps the most difficult issue to address in building simulations. It is the most error-prone area in implementing numerical integration algorithms, especially where discrete and continuous models are present. A well-known and experienced simulationist, Crenshaw [4], says, "As obvious as the point may be, getting x (states) out of sync with t (time) is the most common error in implementations of numerical integration algorithms, and I can't tell you how many times I've seen this mistake made." This mis-synchronization is even made more insidious in that, except for the most simple of systems, it is hard to recognize and discern in the (incorrect) simulation results.

To illustrate the potential pitfall of mixing a discrete operation with a continuous operation, consider the very simple system in *Figure 11*.
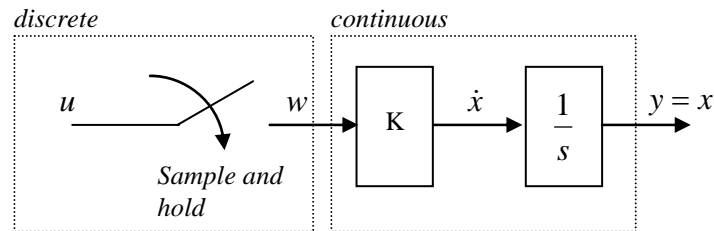
Figure 11. Simple Discrete/Continuous Hybrid System

The sampler, running at some data rate, is a discrete operation and the integrator that follows is continuous. For proper operation and synchronization, the derivative $\dot{x}$ must be updated by the current value of *w*. The state *x* is then updated for the output *y*. The seemingly simple solution is

to update all the sample operations in the simulation before defining (updating) derivatives in this sequence:

Update Discrete Model
$w = u$

Update Continuous Model
$\dot{x} = Kw$

$y = x$

Clearly, the discrete model calculations need to be performed first so that $\dot{x}$ is defined based on the most current value of w. This way, everything gets performed in the proper order. Thus, if all the discrete calculations were performed in one block of code in the integrating time-loop and the continuous calculations in another, the discrete update block of code should be called first.

More generally, this strategy could be implemented in a simulation as shown in **Figure 12**. An additional "update" is added to the integration time-loop by having two dedicated methods for updating both the discretes and continuous models. Updates occur in two phases: first, all the discretes are updated followed by update of the continuous models.
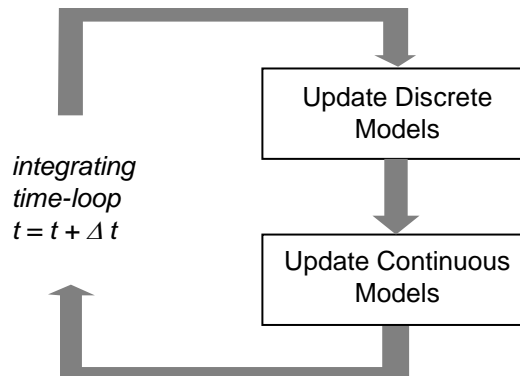
Figure 12. Architecture for Discrete and Continuous Models

The series of operations in the models are slightly rearranged in **Figure 13**. Here, the *sample* function is simply used to report the value of *y* at periodic intervals (as in the OSK *rpt* method).
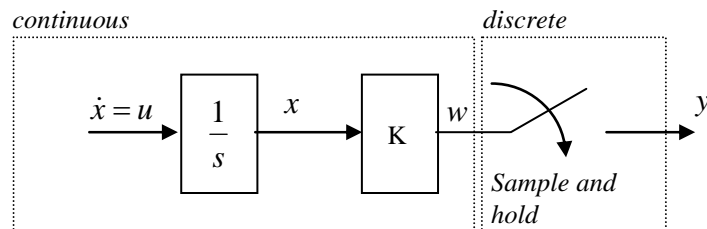
Figure 13. Hybrid System with Operations in Different Order

Let's write the model equations using the update discrete and then update continuous sequence used earlier.

Update Discrete Model
$y = w$

Update Continuous Model
$\dot{x} = u$

$w = kx$

Clearly, there is a synchronization problem here. Executing the discrete model first causes y to be updated before the most current value of w is defined. Again, the solution is seemingly simple – update all the continuous derivatives before the discrete operations.

But we can't have it both ways, the update operations must be performed sequentially so one or the other must be done first.

Special Note:
The careful simulationist will recognize these situations. This synchronization problem can be dealt with, even in an integrating time-loop where either *all* the discrete or continuous updates are performed separately at once. However, the code to implement correct synchronization becomes very cumbersome in many circumstances.

Also, for those modelers who are not so cognizant, lags can be introduced (by simply putting the discrete and continuous operations in their designated code blocks) without this "artificial lag" being readily apparent in the code.

The *sample* function gives us the very powerful capability to precisely place blocks of code for discrete calculations *anywhere* within the code that updates the values for continuous derivatives (if there is any). Hence, there is only the need to call one *update* method in the integrating time-loop. Discrete variable updates are then very easily placed within the single *update* method so that all variables depending on discrete variables are updated with the most current values.

For example, use of *sample* for the system in ***Figure 11*** is

```
if( sample( dt)) {
  w = u;
}
xdot = kw;
y = x;
```

The sample is taken before the value of the derivative is updated. This is the correct order.

Likewise, use of the *sample* function for the slightly modified system in ***Figure 13*** is

```
xdot = u;
w = kx;
if( sample( 1.0)) {
  y = w;
}
```

The power of the *sample* function to precisely specify where discretes occur relative to continuous operations is clearly shown in this simple example. While it is simple to see that a data lag is present under the constraint of performing all discrete operations separately from continuous operations in this example, it might not be so apparent in larger, more complex models and systems. The *sample* function makes it easy to prescribe the precise order that discrete/continuous operations occur.

Special Note:
The *sample* API has proven a successful architecture mechanism for mixing discrete and continuous models and has proven itself over-and-over in many applications [5][6][10].

Next, *sample* is used to schedule an event in the missile/autopilot model.

4.3 Scheduling Events

One-time events are another "bread-and-butter" simulation capability. In concert with the philosophy "keep things simple," this capability is easily applied using a special form of the *sample* method.

The event-generation will be illustrated by adding a gain scheduling capability to our model. The modified system is shown in ***Figure 14***.
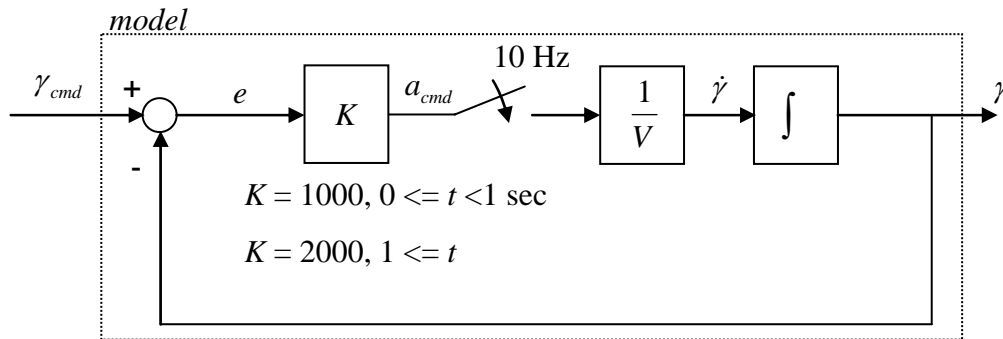
Figure 14. Autopilot with Gain Scheduler

The gain, *K*, is adjusted at t = 1 second from 1000 to 2000.

An alternate second form of the *sample* function is used to specify discrete events:

static int sample(State::EVENT, double t_event);

The function is a boolean function as before, but this time the first argument is a pre-defined constant *State::EVENT*, and the second argument, *t_event*, is the time at which the event is to occur. Again, *sample* returns true if (1) it is at the beginning of an integration time-step, and (2) *t_event* is the current time.

This makes it very easy to implement the gain scheduler in code. The code fragment from model.cpp is

```
0 --------void Model::update() {
1 --------   gamma_cmd = 1.0;
2 --------   v = 1000.0;
3 --------   if( State::sample( State::EVENT, 1.0)) {
4 --------      k = 2000;
5 --------      cout << " EVENT " << State::t << " " << k << endl;
6 --------   }
7 --------   if( State::sample( 0.1)) {
8 --------      a_cmd = k * ( gamma_cmd - gamma);
9 --------      cout << " sample " << State::t << " " << a_cmd << endl;
10-------   }
11-------   gammad = a_cmd / v;
12-------}
```

Everything is the same as before except that we add an additional *if* block[3-6] to change the gain, *K*. An informative *cout* statement[5] is included to confirm that the change has taken place in the printed output.

The model is compiled and linked with as before

dos>make
lin>make

and ran

dos>main
lin>main

```
13-------starting Model...
14-------   sample 0 1000
15-------     0.000 0.000000
16-------   sample 0.1 900
17-------     0.100 0.100000
18-------   sample 0.2 810
19-------     0.200 0.190000
  ...
20-------   sample 0.8 430.467
21-------     0.800 0.569533
22-------   sample 0.9 387.42
23-------     0.900 0.612580
24-------   EVENT 1 2000
25-------   sample 1 697.357
26-------     1.000 0.651322
27-------   sample 1.1 557.886
```

28------- 1.100 0.721057

 ...
29------- sample 1.9 93.5977
30------- 1.900 0.953201
31------- sample 2 74.8781
32------- 2.000 0.962561
33------- 2.000 0.962561

Indeed the gain was changed to 2000 at t = 1 second [24]. Of course, the final state, *gamma*, is different at t = 2 seconds from the previous example due to the larger gain. This also confirms the gain was switched.

Again, this illustrates that more than one *sample* function can be included in any model's *update* method. Here, two were required: one to trigger the gain switch and another to perform the discrete sample. There is no limit to how many *sample* functions can be included – put as many as necessary to represent the model being implemented.

4.4 Building and Connecting Multiple Models

Up until now, we have been working with one single model that is called in the train-of-objects in the main program. This is fine for relatively simple systems, but more complex systems will be decomposed into a series of models. In addition to having more than one system of models in our train-of-objects, multiple objects also introduce the necessity for communicating variables between models. Both of these aspects are illustrated in this example.

Start by using the same system in the previous examples, but this time segregating the autopilot functionality from the missile airframe dynamics model as shown in *Figure 15*.
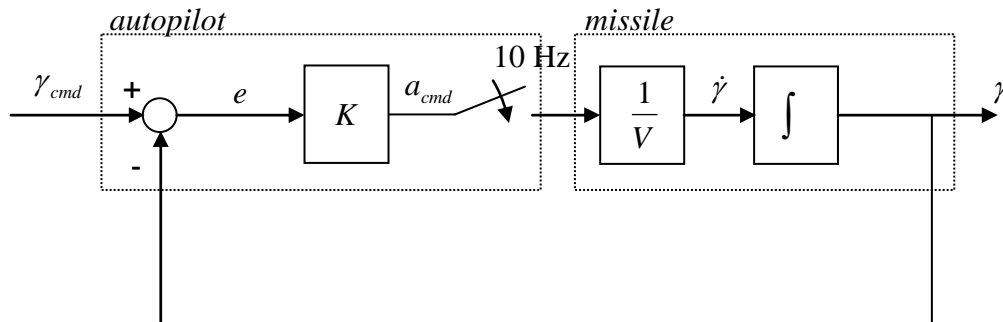
Figure 15. Example System Decomposed into Two Separate Models

Conceptually, this decomposition represents separating the missile's control functions, typically implemented in software on a digital computer, from the physics (airframe dynamics). Separating algorithmic models from "truth" models is certainly a fundamental decomposition in any larger-scale simulation.

4.4.1 Building the Train-of-Objects

First, let's look at how multiple models are programmed in the train-of-objects. The OSK provides the mechanism for defining multiple models in the main program, main.cpp:

```
0 --------#include "autopilot.h"
1 --------#include "missile.h"

2 --------int main() {
3 --------   double  tmax = 2.00;
4 --------   double  dt = 0.01;

5 --------   Autopilot *autopilot = new Autopilot( 1000.0);
6 --------   Missile *missile = new Missile( 0.0);

7 --------   autopilot->getsFrom( missile);
8 --------   missile->getsFrom( autopilot);

9 --------   vector<Block*> vObj0;
10-------   vObj0.push_back( autopilot);
11-------   vObj0.push_back( missile);

12-------   vector< vector<Block*> > vStage;
13-------   vStage.push_back( vObj0);

14-------   double dts[] = { dt};
15-------   Sim *sim = new Sim( dts, tmax, vStage);
16-------   sim->run();
17-------}
```

For now we will assume that separate classes have been defined for the missile and autopilot in *Figure 15*. We will discuss the details of the construction of those classes in a moment. Conceptually, everything is the same as before, but this time we will be adding *two* models to the first stage in the train-of-objects. We will still just have one stage.

The first step is to create the *autopilot* [5] and *missile* [6] objects using their constructors. The autopilot's gain, *K*, is used to define the *autopilot* object [5]. The initial flight path angle, *gamma* [6], is passed to create the *missile* object.

Let's go ahead and build the train-of-objects first and then come back to linking the objects [7,8]. Stage 1 is called *vObj0* [9]. We load its vector with the autopilot and then the missile objects [10,11]. Objects are executed in the order that they are added to the vector in a first-in, first-to-execute manner.

The stage of objects, *vObj0*, is then added [13] as the first and only stage in *vStage* [12]. The stage vector, *vStage*, is then used to create a *sim* object [15].

We know that the autopilot model will need data from the missile model and, conversely, the missile model will need to access variables in the autopilot model. These interconnections are established with the *getsFrom* function [7,8]. The origin of the *getsFrom* function is discussed next, but before proceeding, note how intuitive and readable the code is regarding which models

are connected to which. The code, using the *getsFrom* API, is very close to a literal written statement on how the models are connected.

4.4.2 Linking the Models

As mentioned earlier, a key issue in multiple models is not only providing a mechanism for organizing and executing the models (shown in the previous section), but also how to interconnect the models so that they can exchange data. In this case, the autopilot needs the current flight path angle, γ, from the missile model, and the missile model uses the acceleration, a_{cmd}, commanded by the autopilot. The OSK provides code mechanisms to expedite making these connections.

Connecting models with the OSK involves first defining the object to be connected to, and then providing an access function for retrieving each variable. Let's look at the autopilot model first and its header file, autopilot.h:

```
18-------class Missile;
19-------class Autopilot : public Block {
20------- public:
21-------   Autopilot( double k);
22-------   void getsFrom( Missile *obj) {
23-------     this->missile = obj;
24-------   };
25-------   ACCESS_FN( double, a_cmd);
26------- protected:
27-------   void init();
28-------   void update();
29-------   void rpt();
30-------   double gamma_cmd;
31-------   double k;
32-------   double a_cmd;
33-------   Missile *missile;
34-------};
```

First, provisions must be made to designate which object to connect to (in this case an object created from the *Missile* class). This is done by declaring, as a local variable, a pointer to a *Missile* class object [33]. At this point, the missile class has not been defined, so we simply forward declare the *Missile* class [18] which is promising the compiler that it will be defined later in another compilation unit. A *missile* object is passed to the autopilot's *getsFrom* function [22] and the local missile object pointer is set [23]. This methodology of defining local pointers to external objects and initializing them in a *getsFrom* method is the preferred method for connecting objects. Multiple local objects can be defined, passed to *getsFrom*, and initialized in the case the input for a model will come from multiple sources. This will be illustrated in a large scale example that follows.

That sets up how the autopilot *gets* the parameters it needs. How will other models *supply* any autopilot variables that are needed? Access functions are the preferred way for making internal variables available to other models. The OSK provides a special macro that expedites the definition of access functions. The *ACCESS_FN* [25] macro creates a function that returns the variable named in the second argument. So that any type of variable may be accessed, the type

of the variable returned is specified with the first argument. The name of the access function created by the macro is the variable name appended with an underscore, "_". In this case, an access function called a_cmd_ [25] is created that can be used externally. In another model, suppose a pointer to an autopilot object has been established and a local variable for the acceleration, *a*, has been declared. The acceleration command is accessed using the syntax

a = autopilot->a_cmd_();

Special Note:

Why not just access each models variables directly? While this is possible, direct variable access would allow the accessing model to also *change* the variable being accessed. This might cause hard-to-debug problems in a large simulation where multiple models are being used. Access functions, which simply return the value of a variable, are used so that external models can read but can't change a model's variables, thus eliminating this as a possibility in the event of a malfunction.

Similar steps are taken within the missile model to complete the connections in the missile model's header file, missile.h:

```
35-------class Autopilot;
36-------class Missile : public Block {
37-------  public:
38-------    Missile( double gamma0);
39-------    void getsFrom( Autopilot *obj) {
40-------      this->autopilot = obj;
41-------    };
42-------    ACCESS_FN( double, gamma);
43-------  protected:
44-------    void init();
45-------    void update();
46-------    void rpt();
47-------    double v;
48-------    double gamma0, gamma, gammad;
49-------    Autopilot *autopilot;
50-------};
```

The Autopilot class is forward declared [35] informing the compiler its definition will follow. This enables a pointer to be declared to an *autopilot* object [49]. The *getsFrom* function is defined [39] to provide a means for assigning this autopilot pointer in the main program. The *ACCESS_FN* macro [42] provides an easy means for other models (in this case, an *autopilot* object) to reference the local variable *gamma* through an access method *gamma_*.

Using these established connections in the models themselves is very straightforward. A code fragment from autopilot.cpp shows how the flight path angle from the missile is accessed:

```
51-------void Autopilot::update() {
52-------  if( State::sample( 0.1)) {
53-------    a_cmd = k * ( gamma_cmd - missile->gamma_());
54-------  }
55-------}
```

From the block diagram in *Figure 15*, the autopilot's function is to calculate an acceleration command, *a_cmd*, based on the difference between the commanded flight path angle (which is an internal, local variable) and the current flight angle (which must be obtained from the missile model)[53]. The missile object provides the access function, *gamma_*, for getting the current value of the flight path angle, *gamma*. This access function was the one we defined in the missile's header file, missile.h, using the *ACCESS_FN* macro.

The missile model defines the flight path angle rate derivative based on the commanded acceleration from the autopilot in this code fragment from missile.cpp

```
56-------Missile::Missile( double gamma0_) {
57-------  gamma0 = gamma0_;
58-------  addIntegrator( gamma, gammad);
59-------}
…
60-------void Missile::update() {
61-------  gammad = autopilot->a_cmd_() / v;
62-------}
```

A new state is created for *gamma* in the constructor with the *addIntegrator* method [58]. The state derivative is defined using the commanded acceleration from the autopilot accessed with the function *a_cmd_*[61]. This access function was the one we defined in the autopilot's header file, autopilot.h, using the *ACCESS_FN* macro.

4.4.3 Running the Simulation

To illustrate both stages functioning, a couple of print statements were added to each model's *rpt* method. Code for the *rpt* methods in autopilot.cpp[65] and missile.cpp[71], respectively, produce informative printout so that each model's execution can be observed:

```
63-------void Autopilot::rpt() {
64-------  if( State::sample( 0.1)) {
65-------    printf( "Autopilot %8.3f %8.6f\n", State::t,
66-------          this->a_cmd);
67-------  }
68-------}

69-------void Missile::rpt() {
70-------  if( State::sample( 0.1)) {
71-------    printf( "Missile %8.3f %8.6f\n", State::t,
72-------          this->gamma);
73-------  }
74-------}
```

The model is compiled and linked

dos>make
lin>make

and ran

dos>main
lin>main

The resulting output is

75-------Autopilot   0.000 1000.000000
76-------Missile   0.000 0.000000
77-------Autopilot   0.100 900.000000
78-------Missile   0.100 0.100000
79-------Autopilot   0.200 810.000000
80-------Missile   0.200 0.190000
81-------Autopilot   0.300 729.000000
82-------Missile   0.300 0.271000
83-------Autopilot   0.400 656.100000
84-------Missile   0.400 0.343900
85-------Autopilot   0.500 590.490000
86-------Missile   0.500 0.409510
87-------Autopilot   0.600 531.441000
...
88-------Missile   1.800 0.849905
89-------Autopilot   1.900 135.085172
90-------Missile   1.900 0.864915
91-------Autopilot   2.000 121.576655
92-------Missile   2.000 0.878423
93-------Autopilot   2.000 121.576655
94-------Missile   2.000 0.878423

The models clearly execute sequentially in the desired order at each time-step. Recall that the autopilot model was loaded into the stage 1 vector followed by the missile model. Therefore, the autopilot model executes first at each time-step [75,76]. Otherwise, the numerical results are identical to example 2 (4.2) since the mathematical representation of the system is identical, with the only exception being that here the systems representation was decomposed into two models.

The train-of-objects architecture for multiple models illustrated in this example, along with its provisions for connecting the models, is fully scalable to larger, more complex systems. Within a stage, more models are added by appending them to the STL vector container with the push_back API. Likewise, an arbitrarily large number of connections between models can be made by first declaring the objects to connect with using the *getsFrom* API, and then making variables available with the *ACCESS_FN* macro. This is fully demonstrated in a larger example in Section 1.

In the vein of simulating more complex systems, the next topic to be addressed is creating multiple stages. This is described in the next example.

4.5 Using Multiple Stages (Changing Model Topology)

One of the most challenging features to be addressed in a simulation architecture is providing a mechanism to accommodate different sequences of model execution; in a more concise statement, the simulation must accommodate changes in the *topology* of the system as it evolves in time. On top of that, this change in topology may be in response to events generated within the models themselves as they execute and not known a priori. As the execution sequence of the models change, some may be new models, some may be models in the current sequence that continue propagating from their previous states, or some may even be models in the prior sequence that need to be reinitialized. For example, stages on a missile may be represented by successive, but different models representing changes in aerodynamics, guidance algorithms, or control hardware. Another example might be the processes, and their associated models, required to manufacture a chemical that change at different points in its manufacture.

The OSK provides a concise, effective mechanism to address all of these operational requirements that is easily scaled to complex systems. The system used in the previous examples is extended in this section to illustrate this capability in detail. A larger scale example is shown in the next chapter for a more complex system.

4.5.1 Setting Up the Models

Let's extend the missile steering dynamics example we have been working on to "fly" in two distinct configurations or stages – the first where the missile flies without any control and then flying with the autopilot closing the loop to control the airframe dynamics. Conceptually this might model, at an admittedly somewhat abstract level, a two-stage missile flying out ballistically (no control) with the first stage and then a separating second stage flies under control of the autopilot. The math models representing the two stages are shown in *Figures 16 and 17*.
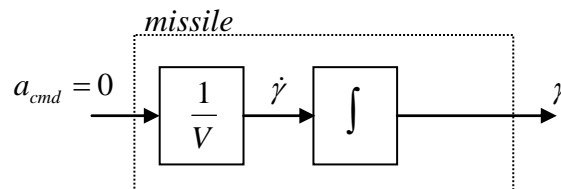
Figure 16. Stage 1 – Missile Flies Open-Loop

Stage 1 is the simple airframe dynamics model we have used all along with zero commanded acceleration.
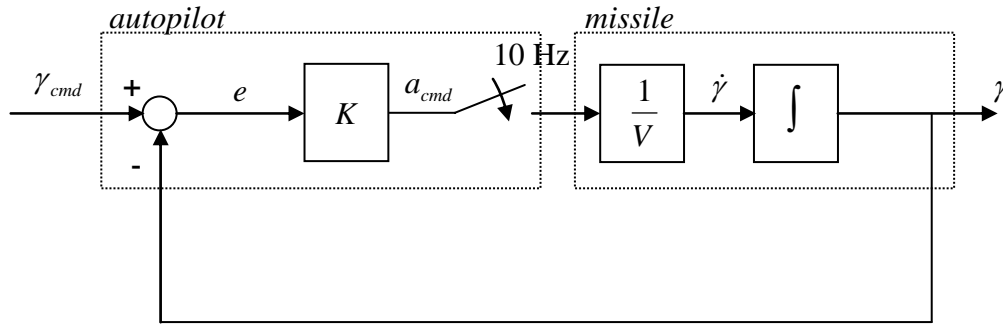
Figure 17. Stage 2 – Missile Flies Closed-Loop

Stage 2 is identical to the model we implemented in Section 4.2. The simulation should start with the stage 1 model and then at some time transition to the stage 2 models. The missile model in stages 1 and 2 is the same model – it simply continues execution from its final state in stage 1 during stage 2.

Building this simulation involves loading the correct model sequence into the train-of-objects, controlling how the models are initialized at the start of each stage, and adding code to trigger the staging event. These steps are described in the following sections.

4.5.2 Loading the Train-of-Objects

The first step is to load the train-of-objects that is passed to the *Sim* class constructor in main.cpp.

```
0 --------int main() {
1 --------   double  tmax = 3.00;
2 --------   double  dt = 0.01;

3 --------   Autopilot *autopilot = new Autopilot( 1000.0);
4 --------   Missile *missile = new Missile( 0.0);

5 --------   autopilot->getsFrom( missile);
6 --------   missile->getsFrom( autopilot);

7 --------   vector<Block*> vObj0;
8 --------  vObj0.push_back( missile);

9 --------   vector<Block*> vObj1;
10-------   vObj1.push_back( autopilot);
11-------   vObj1.push_back( missile);

12-------   vector< vector<Block*> > vStage;
13-------  vStage.push_back( vObj0);
14-------   vStage.push_back( vObj1);

15-------   double dts[] = { dt, dt};
16-------   Sim *sim = new Sim( dts, tmax, vStage);
17-------   sim->run();
18-------}
```

The *autopilot* and *missile* objects are created as before using their respective constructors[3,4]. Connections to the models are established using the objects' *getsFrom* methods[5,6]. The *missile* object is the only model in the first stage[8], *vObj0*. Both the *autopilot* and the *missile* objects are used on the second stage[10,11], *vObj1*. Both stages are used[12-14] to complete the train-of-objects, *vStage*. Note that the missile model used in both stages is one-and-the-same model. It will simply continue executing during the second stage.

4.5.3 Controlling Model Initialization

Using a model in more than one stage necessitates specifying how it is to be initialized at the start of each stage. Should it simply keep executing from its last state in the previous stage? Should it be restarted (reinitialized)? Or should it even be reinitialized based on parameters calculated in the previous stage? The OSK provides a simple, flexible mechanism to address all of these possibilities.

Model initialization takes place in its overridden *init* method. The *Missile* class *init* method is

```
19-------void Missile::init() {
20-------  if( initCount == 0) {
21-------    gamma = gamma0;
22-------    v = 1000.0;
23-------    istage = 0;
24-------  } else if( initCount == 1) {
25-------    istage = 1;
26-------  }
27-------}
```

A special instance variable, *initCount*, is maintained by the kernel for each object. This was introduced in the first example 4.1. The OSK initially sets *initCount* = 0 for each object instance. The *initCount* for that model is then incremented by 1 each time the model's *init* method is called (i.e. at the start of each stage that the model is used in). Thus, *initCount* provides a means for the model itself to know where in the train-of-objects it is being executed.

In this case, a local variable, *istage*[23,25], is used to store the current stage that the model is in between model method calls. Note how *initCount* is used to set *istage*[20-26].

Again, the *initCount* instance variable set by the OSK provides the means to tailor the initialization of each object depending on which stage it is in.

4.5.4 Specifying Transition to Next Stage

Perhaps the most important task in staging is actually setting the conditions and signal for the staging event to occur. Since the missile object is active in both stages, it is the logical place to poll for and trigger the event. The polling should take place in the *update* method since it is called inside the integrating time-loop.

```
28-------void Missile::update() {
29-------  if( State::sample( State::EVENT, 1.00)) {
30-------    Sim::stop = 1;
31-------  }
```

```
32 ------- if( istage == 0) {
33 -------   gammad = 0.0;
34 ------- } else if( istage == 1) {
35 -------   gammad = autopilot->a_cmd_() / v;
36 ------- }
37 -------}
```

Since the staging signal is a one-time event, the *sample* function with an *EVENT* calling argument is used to test for the event[29]. The *sample* function returns true at t = 1 second (the desired staging time).

The *Sim* class variable stop is set to 1[30] to trigger the transition to the next stage in the train-of-objects. Changing the value of *Sim::stop* is a command to go to the next list of objects in *vStage*. *Sim::stop* is automatically initialized to 0 at the start of the simulation.

Setting *Sim::stop* < 0 terminates the simulation at the end of the current integrating step regardless if any stages follow.

Recall the missile uses the state derivative *gammad* and *gamma* as shown in ***Figure 16 and Figure 17*** ($\dot{\gamma}$ and γ). During the first stage, the missile flies open-loop and *gammad* is zero[33]. During the second stage, *gammad* is based on the commanded acceleration from the autopilot, which is obtained from the autopilot's access function, *a_cmd_*[35]. Note how *istage*[32,34] (which is set according to *initCount*) is used to switch between the two definitions for *gammad* based on the current stage.

Together, the kernel variables *Sim::stop* (which you set) and *initCount* (which is set by the kernel) provide the capability to implement a wide variety of object behaviors when they are used in multiple stages.

4.5.5 Running the Simulation

Informative printout is included in each model's *rpt* methods to show the staging process.

In autopilot.cpp:

```
38 -------void Autopilot::rpt() {
39 ------- if( State::sample( 0.1)) {
40 -------   printf( "Autopilot %8.3f %8.6f\n", State::t, this->a_cmd);
41 ------- }
42 -------}
```

In missile.cpp:

```
43 -------void Missile::rpt() {
44 ------- if( State::sample( 0.1)) {
45 -------   printf( "Missile %8.3f %8.6f\n", State::t, this->gamma);
46 ------- }
47 -------}
```

Of course, each model's *rpt* method will only execute if that model is in the current stage.

The program is compiled, linked, and ran:

dos>make
lin>make

and running it

dos>main
lin>main

producing these printed results

```
48-------Missile    0.000 0.000000
49-------Missile    0.100 0.000000
50-------Missile    0.200 0.000000
51-------Missile    0.300 0.000000
52-------Missile    0.400 0.000000
53-------Missile    0.500 0.000000
54-------Missile    0.600 0.000000
55-------Missile    0.700 0.000000
56-------Missile    0.800 0.000000
57-------Missile    0.900 0.000000
58-------Missile    1.000 0.000000
59-------Autopilot    1.000 1000.000000
60-------Missile    1.000 0.000000
61-------Autopilot    1.100 900.000000
62-------Missile    1.100 0.100000
...
63-------Autopilot    2.900 135.085172
64-------Missile    2.900 0.864915
65-------Autopilot    3.000 121.576655
66-------Missile    3.000 0.878423
67-------Autopilot    3.000 121.576655
68-------Missile    3.000 0.878423
```

Only the missile object is executing during stage 1 for t < 1.0 second [48-58]. The stage 2 objects begins execution at t = 1.0 second [58,59]. In stage 2, the autopilot model executes first and then the missile model according to the order that they were added to the list of objects for stage 2. Note that the missile object in stage 2 (which is one-and-the-same object as in stage 1) keeps on executing from its previous state in stage 1.

While this example only had two stages and each stage had only a trivial number of objects, building simulations with numerous stages and models within each stage is easily accomplished using the same methodology used in this example. More stages and models are added simply by making their list of objects in the train-of-objects larger. A more complex example illustrating this scalability follows in the next section.

4.5.6 Capturing Model Data at Stage Transitions

One final issue that needs to be addressed in staging is the provision to observe the staging event itself in output. This is done implicitly if a staging event occurs on an even *sample* function

increment in a model's *rpt* method. What if a staging event occurs at a time other than an even sample increment? How can results (i.e. the object model states) be reported in this case?

The OSK provides two very useful class variables for reporting output that are designed to be used in conjunction with *sample* – *State::tickfirst* and *State::ticklast*. *State::tickfirst* is true if the object is at the start of the integrating time-step and executing for the first time. It is useful for printing out the starting state of an object even if the object does not start operation on an even sample printout increment. *State::ticklast* is true if the object is at the start of the integrating time-step and the simulation has terminated. Again, it is useful for obtaining printout even if the simulation does not terminate on an even sample printout increment.

Special Note:
Why is *State::ticklast* true only at simulation termination instead of at the end of each stage? The values of the state variables and their dependent variables at the beginning of any stage pick up where they left off at the end of the previous stage and, hence, are identical. So using *State::tickfirst* captures the states as they were at the conclusion of the previous stage.

The *rpt* method in missile.cpp can be slightly modified to illustrate how these OSK variables are used to capture output at the stage transition times.

```
69-------void Missile::rpt() {
70-------  if( State::sample( 0.1) || State::tickfirst || State::ticklast) {
71-------    printf( "Missile %8.3f %8.6f\n", State::t, this->gamma);
72-------  }
73-------}
```

State::tickfirst and *State::ticklast* are inserted as *or* conditions[70] in addition to the *sample* function boolean that captures output every 0.1 second.

Let's change the staging time from 1.0 second to 1.250 seconds (which is not an even *sample* increment [70]) to observe operation of *tickfirst*. The staging time is changed by resetting the second argument to *sample* in missile.cpp [75].

```
74-------void Missile::update() {
75-------  if( State::sample( State::EVENT, 1.25)) {
76-------    Sim::stop = 1;
77-------  }
78-------  if( istage == 0) {
79-------    gammad = 0.0;
80-------  } else if( istage == 1) {
81-------    gammad = autopilot->a_cmd_() / v;
82-------  }
83-------}
```

Likewise, let's change the simulation termination time to 3.05 seconds (again which is not an even *sample* output increment [70]) to observe operation of *ticklast*. The simulation termination time[85] is changed in the main program

```
84-------int main() {
85-------   double  tmax = 3.05;
86-------   double  dt = 0.01;
87-------   …
```

Compiling and running the simulation with these changes yields

```
88-------Missile    0.000 0.000000
89-------Missile    0.100 0.000000
90-------Missile    0.200 0.000000
91-------Missile    0.300 0.000000
...
92-------Missile    1.000 0.000000
93-------Missile    1.100 0.000000
94-------Missile    1.200 0.000000
95-------Missile    1.250 0.000000
96-------Autopilot    1.300 1000.000000
97-------Missile    1.300 0.000000
98-------Autopilot    1.400 900.000000
99-------Missile    1.400 0.100000
...
100 -----Missile    2.800 0.794109
101 -----Autopilot    2.900 185.302019
102 -----Missile    2.900 0.814698
103 -----Autopilot    3.000 166.771817
104 -----Missile    3.000 0.833228
105 -----Missile    3.050 0.841567
```

The transition to stage 2 occurs at t = 1.25 seconds and is captured in printout[95] even though it doesn't occur at an even increment of the *sample* output increment (every 0.1 second). The class variable *State::tickfirst* is true when the stage 2 objects begin to execute and, since *tickfirst* is *or'ed* in the *if* statement that polls for printout in the missile object's *rpt* method[70], the *cout* statement is executed.

Also, the final values of the missile parameters are printed[105] at simulation termination even though the simulation's ending time doesn't fall on an even *sample* output increment. The class variable *State::ticklast* is true upon completion of the simulation's final time-step and, since *ticklast* is *or'ed* in the *if* statement that polls for printout in the missile object's *rpt* method[70], the *cout* statement is executed.

Reporting of results at critical simulation events is an important and useful capability that can be easily coded as shown in this example.

4.6 Asynchronous Scheduling

Up until now, we have used the OSK's *sample* function for two purposes:

(1) Periodic sampling
```
if( State::sample( 0.01)) {
  ... do some discretes here every 0.1 sec ...
}
```

(2) Triggering events
```
if(State:: sample( State::EVENT, 1.5)) {
  ... do some event at t = 1.5 sec ...
}
```

An advanced, but frequently needed, simulation capability is to generate periodic events that are not exactly periodic. An example of this is some sampling process that "drifts" with time. An associated capability is the need to generate an event at any arbitrary time. The OSK *sample* function, which has the very simple API already described, is remarkably powerful and can also perform these two capabilities. The remarkable aspect is that this is accomplished largely transparent to the user using the same API that already exists.

The second form of the *sample* function has an asynchronous feature – the event time passed as the second argument can be *any* arbitrary time that is not restricted to an even increment of the integrating time-step. The OSK automatically adjusts its time-stepping to accommodate the event time within its periodic stepping of the integrating time increment.

This section presents four examples illustrating how to extend this capability beyond the simple event *sample* in Section 4.3. These are prefaced by a discussion of asynchronous scheduling and its application. We will then look at how to put these capabilities to use in a variety of situations.

4.6.1 Description

The OSK implements a full *runtime dynamic asynchronous* scheduling capability. What does this mean? At a top-level, this provides the means to implement very sophisticated simulation functionalities to address a wide range of sampled data behaviors. Let's look in detail at what this capability entails.

4.6.1.1   Runtime Scheduling

Scheduled events can be added or deleted during runtime. This also includes the possibility that the time of the event may not be known a priori – the time of the event is the result of calculations done when the simulation is running. Of course, the event times may also be constants at the start of the simulation.

4.6.1.2   Dynamic Scheduling

The time of one specific event may evolve or change with time. That is, once the event time is determined and scheduled, it may change as the result of subsequent, more refined calculations.

The OSK implements a unique scheduling approach. As opposed to storing event times in a queue, future event times are only active if they are actually defined in the currently executing time-step. An event time's definition must be active until the time it occurs. This is easily accomplished by a simple equality statement

t_event = 1.314 + t0;
if(State::sample(State::EVENT, t_event) {
..

Thus, the time of the event will change if *t0* changes during runtime – no other coding needs to be done to account for this. ***Figure 18*** illustrates how this works in a general sense.
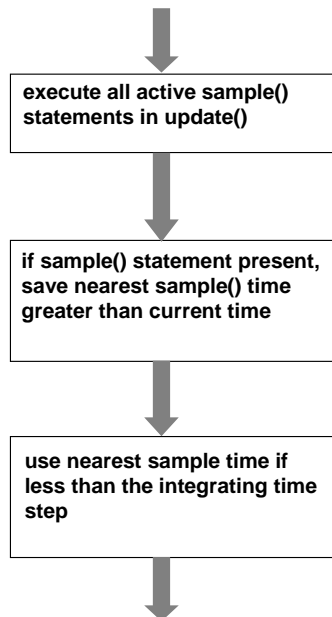
Figure 18. Dynamic Scheduling Logic

The OSK simply looks forward from its current time to the nearest time to step to, be it the next integrating time-step or an event time that was defined by *sample* during the current integrating time-step. An example is shown in ***Figure 19***.

these statements executed in models' update() methods

```
if( State::sample( 0.2) { ...
```

```
if( State::sample( State::EVENT, 0.05) { ...
```

```
if( State::sample( State::EVENT, 0.32) { ...
```

(integrating dt = 0.5 sec)

current time
sample EVENT 0.05
sample 0.2
sample EVENT 0.32
sample 0.4
integrating dt = 0.5

0.0   0.1   0.2   0.3   0.4   0.5

time

actual integrating dt set to capture all points
dt1 = 0.05 sec (from 0 to 0.05 sec)
dt2 = 0.15 sec (from 0.05 to 0.2 sec)
dt3 = 0.12 sec (from (0.2 to 0.32 sec)
dt4 = 0.08 sec (from 0.32 to 0.4 sec)
dt5 = 0.1 sec (from 0.4 to 0.5 sec)

Figure 19. Numerical Example for OSK Time-Stepping

This approach contrasts with (and is much simpler) than a scheme where event times are stored in a queue. In this case, if an event time changes, its old value must be removed from the queue and its new value added. This concept is illustrated in one of the examples that follow.

4.6.1.3   Asynchronous Scheduling

An event may be scheduled at *any* time with *any* resolution. The event does not necessarily have to occur exactly on an integrating time-step. For instance, in the code fragment

if(State::sample(State::EVENT, 1.03128)) {
  ... do some event at t = 1.03128 sec ...
 }

the event will "fire" at t = 1.03128 seconds regardless of the specified integrating time-step. Also, the event time can be specified without regard to the "fineness" of the integrating time-step. For instance, the event will occur precisely at t = 1.03128 seconds even if the integrating time-step is 1 second. Internally, the OSK steps to 1.01318 seconds from 1 second in the integrating time loop. After executing all the models' *update* methods, the OSK then steps to 2 seconds. All this is done internally and is transparent to you except for the *sample* statement itself.

4.6.2 Syntax

Table 2 summarizes the syntax for the *sample* function.

Table 2. Full Sample Syntax Including Asynchronous Scheduling

Syntax	Description
if(State::sample(double dt)) {...}	Used to schedule and test for a periodically occurring event every dt sec.
if(State::sample(State::EVENT, double t)) {...}	Used to schedule and test for an event defined to occur at t sec. The variable t can change during the simulation to trigger more than one event with a single statement.
State::sample(State::EVENT, double t));	The statement form of *sample*. Used to schedule an event at time t. This will insure the OSK time-steps to this precise time.
if(State::sample()) {...}	Special form of *sample* that returns true if it is the start of an integrating time-step. Useful for debugging or logging exactly where time-steps occur.

Again, it is most important to note that candidate step times are collected via the *sample* method calls executed each *update* cycle – the nearest time greater than the current time is used to determine the next step size. If a *sample* call is not in actively executing code during the current *update* cycle, it is not used in computing the length of the next time-step. This obviates the requirement (and additional user code interface complexity) of adding/removing/changing events in a queue or stack – simply add logic to change the argument to *sample* inside the *update* method.

The final form of *sample* in *Table 2* bears special mention. It is useful during model development to selectively embed print (i.e. *cout*) statements in the code to monitor variables during execution (as we have done in many of the examples to illustrate features of their operation). One problem, for numerical integration algorithms that perform multiple evaluations of the derivatives within a single time-step (which the vast majority of algorithms do), is that simply inserting a print statement would lead to excessive, and probably undesired, output of not only values at the end of the time-step, but also intermediate results. Using the *sample* function in an *if* block with no arguments is a convenient way to "capture" output at the bounds of each integrating time-step and, hence, to fully observe behavior of the variables.

4.6.3 Examples

The features discussed in the previous section are illustrated with some simple examples using the simple system that we started with in the first example in Section 4.1. Each example uses this system as a starting point. Again, only the code that is pertinent to discussion is shown. The full code listing should be consulted for additional insight to operation.

4.6.3.1   Capturing Asynchronous Events

Let's start with a basic example that captures multiple one-time events that are not on integrating time-steps. With an integrating time-step of 0.1 second (set in main.cpp), it is desired to generate events at t = 0.15, 0.185, and 0.1855 second. Note that not only are these event times not even multiples of 0.1, but they are at considerably finer resolution than increments of 0.1 second.

These events are coded using the *sample* function in model.cpp:

```
0 --------void Model::update() {
1 --------   gamma_cmd = 1.0;
2 --------   if( State::sample( State::EVENT, 0.1855)) {
3 --------      k = 1000;
4 --------      cout << " EVENT " << State::t << " " << k << endl;
5 --------   }
6 --------   if( State::sample( State::EVENT, 0.15)) {
7 --------      k = 1000;
8 --------      cout << " EVENT " << State::t << " " << k << endl;
9 --------   }
10-------   if( State::sample( State::EVENT, 0.185)) {
11-------      k = 1000;
12-------      cout << " EVENT " << State::t << " " << k << endl;
13-------   }
14-------   a_cmd = k * ( gamma_cmd - gamma);
15-------   v = 1000.0;
16-------   gammad = a_cmd / v;
17-------}
```

The events are polled for using *sample* in conjunction with *if* statements[2,6,10]. The *sample* statements do not have to appear in any order with respect to the event times. The important thing is that each *sample* function must be executed every integrating time-step (at least until the event is generated). Print statements (*cout*) have been added to confirm the events were indeed generated[4,8,12].

Running the simulation yields these results

```
18-------starting Model...
19-------   0.000   0.1000 0.000000
20-------   0.100   0.0500 0.095163
21-------   EVENT 0.15 1000
22-------   EVENT 0.185 1000
23-------   EVENT 0.1855 1000
24-------   0.200   0.1000 0.181269
25-------   0.300   0.1000 0.259182
26-------   0.400   0.1000 0.329680
```

```
27-------   0.500   0.1000 0.393469
28-------   0.600   0.1000 0.451188
29-------   0.700   0.1000 0.503414
30-------   0.800   0.1000 0.550671
31-------   0.900   0.1000 0.593430
32 -------   1.000   0.1000 0.632120
```

The events did indeed "fire" at the correct times[21-23]. Otherwise, the OSK time-stepped according to the integrating time-step (a *sample* function was used in the *Model* object's *rpt* method to report output every 0.1 second, the integrating time-step).

4.6.3.2   Using sample to Generate Time-Steps

One might make the observation that specifying a constant time-step in defining the train-of-objects in main is too constricting. Regardless of the validity of this observation, *sample* can be used to easily adjust the time-step within a model, as shown in this example.

Suppose in main.cpp the integrating time-step is set to 1 second[2] and this is the integrating time-step used in defining the train-of-objects[12,13]:

```
0 --------int main() {
1 --------   double  tmax = 1.0;
2 --------   double  dt = 1.0;
3 --------
4 --------   Model *model = new Model( 0.0);
5 --------
6 -------   vector<Block*> vObj0;
7 --------   vObj0.push_back( model);
8 --------
9 --------   vector< vector<Block*> > vStage;
10-------   vStage.push_back( vObj0);
11-------
12-------   double dts[] = { dt};
13-------   Sim *sim = new Sim( dts, tmax, vStage);
14-------   sim->run();
15-------}
```

With this relatively coarse time-step in place, let's say it is desired within a model to use a finer time-step. This is easily accomplished with the *sample* function's event generation capability.

First, define and initialize an event time variable, *te*, in model.cpp:

```
16-------void Model::init() {
17-------   cout << "starting Model...\n";
18-------   if( initCount == 0) {
19-------     gamma = gamma0;
20-------     k = 1000.0;
21-------     te = 0.0;
22-------   }
23-------}
```

We have only added the line to initialize *te*[21]. (Of course, *te* has to be declared in the header file, model.h.)

58

Adding a *sample if* block[25-29] to the *update* method in model.cpp effectively forces a finer integrating time-step:

```
24-------void Model::update() {
25-------  if( State::sample( State::EVENT, te)) {
26-------    te += 0.1;
27-------    State::sample( State::EVENT, te);
28-------    cout << "**** sample " << State::t << " " << State::dt << endl;
29-------  }
30-------  gamma_cmd = 1.0;
31-------  a_cmd = k * ( gamma_cmd - gamma);
32-------  v = 1000.0;
33-------  gammad = a_cmd / v;
34-------}
```

First, *sample* is used to generate an event at the current *te*[25]. The *update* method is executed successively over-and-over in the integrating time loop. The *sample* function will cause the integrator to stop at time t = *te*. When, and only when, this happens, *te* is incremented. Now, a statement form of *sample*[27] (as opposed to functional form) is immediately used to schedule the next *sample* time, *te*. Note what would happen in the absence of this statement: the boolean *sample* test may never be true since the simulation may time-step beyond *te* for the next step (since the original *sample* function[25] is not executed until the next time-step, in this case 1 second). We must intervene and immediately inform the OSK of the new event time as it schedules the next time to step to.

An informative print statement[28] is included to observe the event. Also, maybe redundantly, the no-argument form of *sample* is used in the *rpt* method to print at every integrating time-step.

```
35-------void Model::rpt() {
36-------  if( State::sample()) {
37-------    printf( "%8.3f %8.4f %8.6f\n", State::t, State::dt, this->gamma);
38-------  }
39-------}
```

Results are

```
40-------starting Model...
41-------**** sample 0 0.1
42-------   0.000   0.1000 0.000000
43-------**** sample 0.1 0.1
44-------   0.100   0.1000 0.095163
45-------**** sample 0.2 0.1
46-------   0.200   0.1000 0.181269
47-------...
48-------   0.900   0.1000 0.593430
49-------**** sample 1 0.1
50-------   1.000   0.1000 0.632120
```

The printout confirms that the effective integrating time-step was 0.1 second even though it was specified as 1 second when the train-of-objects was created.

4.6.3.3   Using sample to Control the Time-Step

This example is a slightly more complex variation of the previous example and depicts a common simulation functional necessity: enforcing a finer integrating time increment over a time window. This capability is useful where the dynamics of a system suddenly become more active over a limited time and a smaller integrating time-step is required. An example is a missile attitude control motor with a detailed pulse shape that intermittently fires. Between pulses, the dynamics of the missile are slowly changing relative to the energetic dynamics of the motors firing.

For this example, assume the train-of-objects is built with a time-step t = 0.1 second. It is desired to use this integrating time-step except over a short 0.05 second window that begins at t = 0.55 and ends at t = 0.60 second, where a time-step of 0.01 second is to be used. Note that the finer integrating time rate begins at a non-even increment, t = 0.55, of the original time-step, 0.1 second (just to make the problem "harder").

We proceed similarly as before in the previous example – in fact, implementing this additional capability is not much more complex.

First define and initialize an event time variable, *te* in model.cpp:

```
0 --------void Model::init() {
1 --------   cout << "starting Model...\n";
2 --------   if( initCount == 0) {
3 --------     gamma = gamma0;
4 --------     k = 1000.0;
5 --------     te = 0.55;
6 --------   }
7 --------}
```

We have only added the line to initialize *te*[5]. (Of course, *te* has to be declared in the header file, model.h.).

As before, a *sample if* block[9-15] is added to the existing *update* method in model.cpp:

```
8 --------void Model::update() {
9 --------   if( State::sample( State::EVENT, te)) {
10-------     te += 0.01;
11-------     if( te <= 0.6 + EPS) {
12-------       State::sample( State::EVENT, te);
13-------     }
14-------     cout << "**** sample " << State::t << " " << State::dt << endl;
15-------   }
16-------   gamma_cmd = 1.0;
17-------   a_cmd = k * ( gamma_cmd - gamma);
18-------   v = 1000.0;
19-------   gammad = a_cmd / v;
20-------}
```

Code inside the *if* block is not executed until t = *te*$^{<9>}$ (initially 0.55 second). The event time *te* is incremented$^{<10>}$ to the next time inside the window where smaller integrating time-steps are to be taken. As long as the next time is within the window$^{<11>}$, we go ahead and tell the OSK to schedule it with the statement form of *sample*$^{<12>}$. Otherwise, the OSK simply schedules the next step at the nearest integrating time-step originally specified. The variable *EPS*$^{<11>}$ is a very small number and is used to negate any round-off error in the binary representation of *te*.

Remember, the OSK schedules the boundary of the next integration time-step after all the models' *update* methods have been executed. If the statement form of *sample*$^{<12>}$ is not used to immediately tell the OSK a finer step needs to be scheduled, the OSK will jump to the next normal (and coarser) integrating time-step. In this case, the next time the main *sample*$^{<9>}$ function is executed, the time period with the finer step would have already passed and simply ignored.

As before, the statement form of *sample*$^{<22>}$ is used in the *rpt* method in model.cpp to double-confirm the simulation time-stepping:

```
21-------void Model::rpt() {
22-------  if( State::sample()) {
23-------    printf( "%8.3f %8.4f %8.6f\n", State::t, State::dt, this->gamma);
24-------  }
25-------}
```

Executing the simulation:

```
26-------starting Model...
27-------  0.000   0.1000 0.000000
28-------  0.100   0.1000 0.095163
29-------  0.200   0.1000 0.181269
30-------  0.300   0.1000 0.259182
31-------  0.400   0.1000 0.329680
32-------  0.500   0.0500 0.393469
33-------**** sample 0.55 0.01
34-------  0.550   0.0100 0.423050
35-------**** sample 0.56 0.01
36-------  0.560   0.0100 0.428791
37-------**** sample 0.57 0.01
38-------  0.570   0.0100 0.434474
39-------**** sample 0.58 0.01
40-------  0.580   0.0100 0.440101
41-------**** sample 0.59 0.01
42-------  0.590   0.0100 0.445672
43-------**** sample 0.6 0.1
44-------  0.600   0.1000 0.451188
45-------  0.700   0.1000 0.503414
46-------  0.800   0.1000 0.550671
47-------  0.900   0.1000 0.593430
48-------  1.000   0.1000 0.632120
```

The simulation starts using the originally specified time-step$^{<27-32>}$. The finer integrating time-step, enforced by the *sample* function, is active over the time window t = 0.55 second to t = 0.60 second$^{<33-43>}$. After that time, operation according to the original time-step resumes$^{<44-48>}$.

4.6.3.4   Comprehensive Sample Demonstration

This final asynchronous sampling example incorporates several of the features previously demonstrated, this time in a more complex example. Several aspects of the *sample* function are shown illustrating it can be used in an arbitrarily complex simulation. Detailed examination of this example, with several *sample* functions interacting with the kernel, will give insight into the *sample* function's operation in how it controls the time-stepping.

The multi-stage example in Section 4.5, serves as the starting point. Recall, this example consisted of a "two-stage" missile where the first stage was only the missile steering dynamics model and the second stage was a continuation of the missile steering dynamics model receiving acceleration commands from an autopilot model. The transition between the two stages was specified at t = 1 second.

4.6.3.4.1 Model Problem Statement

Let's apply the *sample* function in different ways and in multiple places within the models. These model changes are stated in ***Table 3***.

Table 3. Modifications to Demonstrate Sample Function

#	Modification	Code Unit
1	Default integrating time increment = 0.1 sec to create train-of-objects	main.cpp
2	Start finer integrating time increment (0.01 sec) at t = 1.55 to 1.63 sec during second stage	autopilot.cpp
3	Schedule two very-closely spaced one-time events (1.6002 and 1.6003) at finer resolution than the integrating time-step	autopilot.cpp
4	Schedule autopilot output every 0.2 sec	autopilot.cpp
5	Schedule missile output every time-step (to observe how actual time-stepping occurs)	missile.cpp

All of these capabilities have been demonstrated in the previous examples. These behaviors were selected to exercise a broad range of the *sample* function features in a single simulation.

4.6.3.4.2 Coding

The *sample* function's simple but flexible interface makes it straightforward to implement these relatively complex simulation behaviors. Following is an item-by-item description of the coding for the modifications listed in ***Table 3***.

Modification #1 is implemented by setting the integrating time increment[2] used to create the train-of-objects[7,8] in main.cpp:

```
0 --------int main() {
1 --------   double  tmax = 2.00;
2 --------   double  dt = 0.1;
3 --------
4 --------   Autopilot *autopilot = new Autopilot( 1000.0);
5 --------   Missile *missile = new Missile( 0.0);
6 --------...
7 --------   double dts[] = { dt, dt};
8 --------   Sim *sim = new Sim( dts, tmax, vStage);
9 --------   sim->run();
10-------}
```

Modification #2 is coded by first initializing an event time, *te*[14], in autopilot.cpp

```
11-------void Autopilot::init() {
12-------   if( initCount == 0) {
13-------     gamma_cmd = 1.0;
14-------     te = 1.55;
15-------   }
16-------}
```

A *sample* function in conjunction with an *if* block[22-27] is used to make the time-step finer over the time window $1.55 < t < 1.63$ in autopilot.cpp

```
17-------void Autopilot::update() {
18-------   a_cmd = k * ( gamma_cmd - missile->gamma_());
19-------   // throw these in to see if they get caught by time stepping
20-------   State::sample( State::EVENT, 1.6002);
21-------   State::sample( State::EVENT, 1.6001);
22-------   if( State::sample( State::EVENT, te)) {
23-------     if( te <= 1.63 - EPS) {
24-------       te += 0.01;
25-------     }
26-------     State::sample( State::EVENT, te);
27-------   }
28-------}
```

The two very closely scheduled events for modification #3 are implemented using the statement form of *sample*[20,21].

Autopilot output at every 0.2 second[30] (modification #4) is implemented in the *rpt* method in autopilot.cpp:

```
29-------void Autopilot::rpt() {
30-------   if( State::sample( 0.2)) {
31-------     printf( "Autopilot %6.4f %8.6f\n", State::t, this->a_cmd);
32-------   }
```

Finally, the no-argument form of *sample*[34] is used to code modification #5 in the *rpt* method of missile.cpp:

```
33-------void Missile::rpt() {
34-------  if( State::sample()) {
35-------    printf( "Missile %8.4f %8.4f %8.6f\n", State::t, State::dt, this->gamma);
36-------  }
37-------}
```

This causes output to be reported at the beginning of every integration time-step so that we can monitor the operation of all the event generators in ***Table 3***.

4.6.3.4.3 Running the Simulation

Running the simulation yields:

```
38-------Missile   0.0000   0.1000 0.000000
39-------Missile   0.1000   0.1000 0.000000
40-------Missile   0.2000   0.1000 0.000000
41-------...
42-------Missile   0.8000   0.1000 0.000000
43-------Missile   0.9000   0.1000 0.000000
44-------Missile   1.0000   0.1000 0.000000
45-------Autopilot 1.0000 1000.000000
46-------Missile   1.0000   0.1000 0.000000
47-------Missile   1.1000   0.1000 0.095163
48-------Autopilot 1.2000 818.730901
49-------Missile   1.2000   0.1000 0.181269
50-------Missile   1.3000   0.1000 0.259182
51-------Autopilot 1.4000 670.320289
52-------Missile   1.4000   0.1000 0.329680
53-------Missile   1.5000   0.0500 0.393469
54-------Missile   1.5500   0.0100 0.423050
55-------Missile   1.5600   0.0100 0.428791
56-------Missile   1.5700   0.0100 0.434474
57-------Missile   1.5800   0.0100 0.440101
58-------Missile   1.5900   0.0100 0.445672
59-------Autopilot 1.6000 548.811886
60-------Missile   1.6000   0.0001 0.451188
61-------Missile   1.6001   0.0001 0.451243
62-------Missile   1.6002   0.0098 0.451298
63-------Missile   1.6100   0.0100 0.456649
64-------Missile   1.6200   0.0100 0.462055
65-------Missile   1.6300   0.0700 0.467408
66-------Missile   1.7000   0.1000 0.503414
67-------Autopilot 1.8000 449.329216
68-------Missile   1.8000   0.1000 0.550671
69-------Missile   1.9000   0.1000 0.593430
70-------Autopilot 2.0000 367.879714
71-------Missile   2.0000   0.1000 0.632120
72-------Autopilot 2.0000 367.879714
73-------Missile   2.0000   0.1000 0.632120
```

Only the missile model is executing[38-44] up until t = 1 second. Because the no-argument form of *sample* was used in the missile object's *rpt* method, output occurs at the original integrating time-step[38-44], 0.1 second.

The stage 2 objects begin execution at t = 1 second, including the autopilot[45]. The original integrating time increment is maintained up until t = 1.5 seconds[53]. The autopilot generates an event at t = 1.55 seconds[54] and causes the integrating time-step to change to 0.01 second up until t = 1.63 seconds[65] (modification #2). Again, visibility into the kernel's time-stepping is provided by the no-argument form of *sample* in the missile object's *rpt* method (modification #5). The closely spaced one-time events (modification #3) are observed at t = 1.6001[61] and t = 1.6002[62]. The time-step reverts back to evenly-spaced increments of 0.1 second at the end of the fine time-stepping window[66.68.69.71]. Finally note that throughout the second stage output, the autopilot output is every 0.2 second as specified in its *rpt* method (modification #4).

This example, and the other asynchronous examples, show that a wide range of complex simulation event behaviors can be implemented using the *sample* function to fine-tune and control time-stepping. The *sample* function in its many configurations can be used to implement virtually any strategy for seamless mixing of periodic and asynchronous events. The runtime dynamic asynchronous scheduling capability greatly simplifies the process of adding, removing, or changing events. There is no need for a complex API to keep up with events in a queue or stack.

The *sample* function API makes it easy to avoid synchronization issues associated with mixing continuous and discrete systems, as discussed in Section 4.2.2. Discrete operations are simply inserted in the code in the same order that they are performed in a model. This obviates the necessity for defining separate update blocks in the integrating time-loop code for discrete and continuous models and then having to worry about the time-sequencing interaction between them.

All these advanced simulation capabilities are available in a single, straightforward *sample* function API.

4.7 Summary

The full extent of CMD's OSK functionality has been covered by the examples in this section. Armed with knowledge of the capabilities shown, you should be ready to build simulations of your own with CMD. Any of these examples provide a good "go-by" as a starting place to begin, if desired.

The next section illustrates using these capabilities for a large-scale, more complex simulation.

5   SCALABILITY – BUILDING A 6 DEGREE-OF-FREEDOM (6DOF) SIMULATION

One problem with a lot of simulation documentation is that simple examples are used to illustrate basic concepts and operation. This is a necessity so that the concepts being communicated aren't masked by the details of an overly complex system. For completeness, this section explores techniques of using the concepts in the preceding examples for a large simulation with many stages, many models within the stages, and complex interactions and sequencing between the models. Building a missile Six Degree-Of-Freedom (6DOF) simulation is a good medium to explore this.

Special Note:
The completed code for the 6DOF simulation is included as directory ex_6dof in the distribution.

5.1 The System to be Simulated

This section is not meant to be a tutorial for building 6DOF simulations in general. Other excellent references are available for that purpose [11][12]. To avoid too much operational description that would mask the intention here of describing CMD application to a more complex system, only the details of the models pertinent to the simulation construction are included. Brief descriptions of the models themselves can be found in Appendix 1 for those who wish to investigate further.

Figure 20 illustrates a purely hypothetical missile to simulate. Within its models, all parameters are contrived and represent no particular missile.

Kill Vehicle
- **Jet interaction control**
- **Lateral and roll autopilot**
- **Two-axis gimbal**
- **Homes on ballistic target**
- **Proportional navigation**

Stage 2
- **Ballistic**
- **Time-varying thrust profile**
- **Aero f(alpha, Mach #)**

Stage 1
- **Ballistic**
- **Time-varying thrust profile**
- **Aero f(alpha, Mach #)**
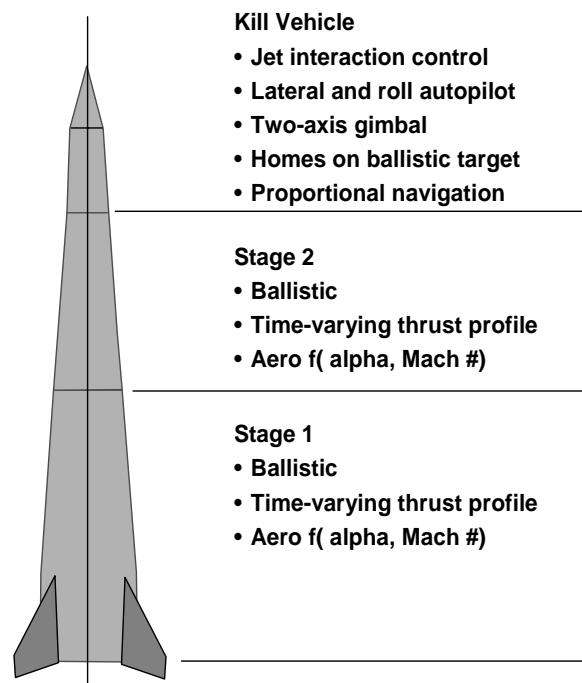
Figure 20. Hypothetical Defense Interceptor Missile

This configuration, and its associated math model formulation, was designed to be sufficiently complex to illustrate aspects of building a 6DOF simulation in CMD. The experienced 6DOF modeler will recognize that the individual models are relatively simple but they could serve as the functional "placeholders" for adding more fidelity.

The missile will engage a target as shown in *Figure 21*.

ballistic re-entry vehicle

- flat earth
- +z axis down

- intercept, t = 14.189 sec
- alt = 15125 m
- miss = 0.064 m

- Stage 2 separation, t = 9.90 sec
- alt = 10928 m
- velocity = 1325.67 m/sec

Note: All parameter values are approximate.

- Stage 1 separation, t = 2.08 sec
- alt = 1211 m
- velocity = 1043 m/sec

- launch at 65 deg elevation
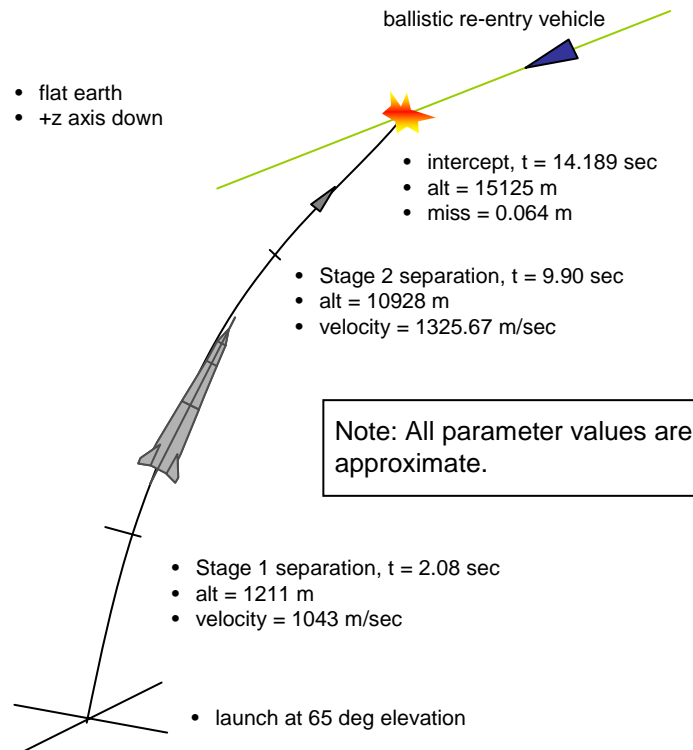
Figure 21. 6DOF Simulation Scenario

The objective is to build a full-up 6DOF simulation of the interceptor missile, starting at launch, and terminating at time-of-closest approach with the target.

5.2 Object-Oriented System Representation

Compared to the example systems illustrated earlier, the complexity of building this simulation seems daunting at first look. However, a working simulation is readily constructed by systematically applying the same principles of the CMD architecture used in the examples.

Given the mathematical formulation of the system – the first question is how to organize these mathematical representations into models. *Figure 22* shows one way to decompose homing missile system functionality. This is based on a much tried-and-tested architecture successfully used on past 6DOF simulations [13]. This serves as the basis to categorize and define the math models. All of the math model formulations are assigned to one of the functional categories in the figure.
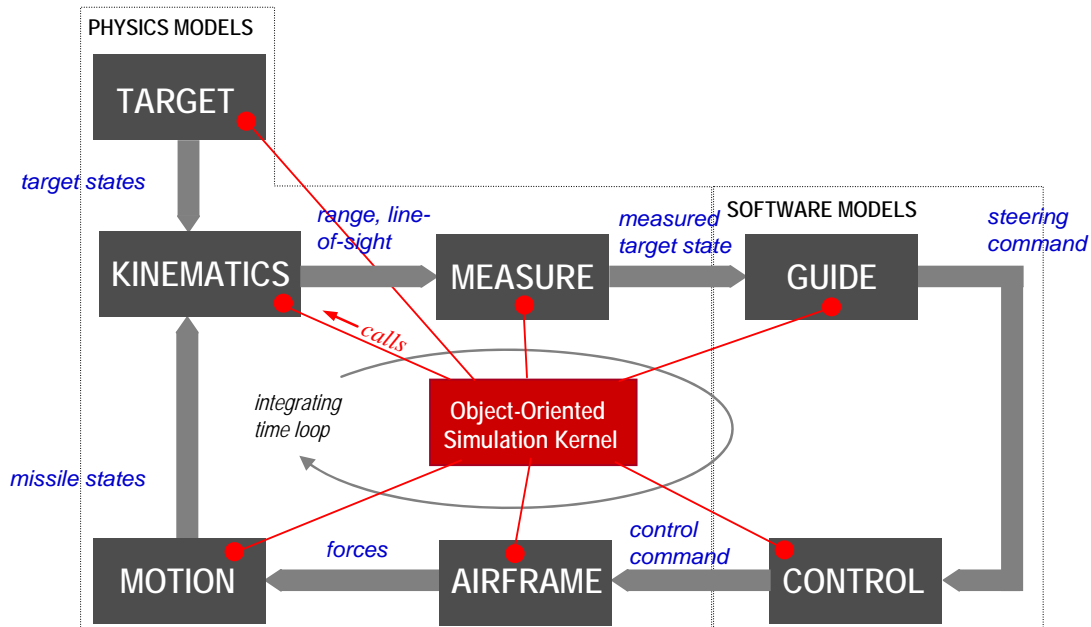
Figure 22. Generalized Homing Missile System Functional Decomposition

It is a seamless mapping in the context of the CMD's O-O architecture to let each block in the
figure represent a class. Construction begins by coding the algorithms for the math models into
the appropriate C++ class. For this example, this has already been done (since the purpose here
is to illustrate the total simulation, not encoding individual models).

These class objects are then collected into a stage in the train-of-objects. Multiple
representations of these series of objects are then used to compose the three stages on the missile.

It is perhaps most clear how to do this in code, so let's proceed directly to that.

5.3 Building the Train-of-Objects

The train-of-objects is built in main.cpp in the following steps. For brevity of presentation, only
code fragments pertinent to the discussion are shown. Of course, the full code listing in the
distribution can be consulted for further examination. All fragments shown here for illustration
are from main.cpp (in the sequential order that they appear in the code).

Some objects will be one-and-the-same for all stages and are created from their class
definitions[3,4,5]:

```
0 --------int main() {
1 --------  ...
2 --------  // same objects for all stages
3 --------  Kinem *kinem      = new Kinem( file_kv, output);
4 --------  Motion *motion    = new Motion( file_kv, output);
5 --------  RVTraj *target     = new RVTraj( file_kv, output);
```

For instance, the equations-of-motion will be the same for all stages where only its inputs change depending on the current stage airframe configuration. Likewise, the target model will be active for the entire flight, as well as a kinematics class that keeps up with the missile's and target's relative states (a relative geometry model). The first argument in the constructor, *file_kv*, is the name of the data file to initialize the object and output is a previously declared object that encapsulates special output data file functions (more on this later).

Next, we create the dedicated stage 1 and stage 2 objects:

```
6 -------- // stage 1 objects
7 -------- Control_boost *control1 = new Control_boost( file_stage1, output);
8 -------- Airframe_boost *airframe1  = new Airframe_boost( file_stage1, output);
9 --------
10------- // stage 2 objects
11------- Control_boost *control2 = new Control_boost( file_stage2, output);
12------- Airframe_boost *airframe2  = new Airframe_boost( file_stage2, output);
```

Note that these objects are created from the same classes[7,11] and [8,12] in stage1 and stage 2. They will be the same in that their math model code is identical but different in that they are using two separate sets of data, *file_stage1* and *file_stage2*.

The final stage, which homes on the target, is the most complex and has four dedicated objects:

```
13-------// kill vehicle objects
14------- Gimbal *gimbal     = new Gimbal( file_kv, output);
15------- Guide *guide       = new Guide( file_kv, output);
16------- Control *control   = new Control( file_kv, output);
17------- Airframe *airframe  = new Airframe( file_kv, output);
```

Now that the model objects have been created, the next step is to define their connectivities so that they may exchange data:

```
18------- // set references
19------- kinem->getsFrom( motion, target);
20------- gimbal->getsFrom( motion, kinem);
21------- guide->getsFrom( gimbal);
22------- control->getsFrom( gimbal, guide, motion);
23------- airframe1->getsFrom( control1, motion, kinem);
24------- airframe2->getsFrom( control2, motion, kinem);
25------- airframe->getsFrom( control, motion, kinem);
26------- motion->getsFrom( airframe1, kinem); // will change at staging event
27------- target->getsFrom();
```

This is done identically as before using each object's *getFrom* method. The objects in the *getsFrom* argument lists are the data sources for the calling object. As seen earlier, *getsFrom* internally assigns pointers that point to the objects' access functions. Note the clarity of the code here – it is very easy to see the models' connectivities.

Now we proceed directly to build the train-of-objects. The stages are built by adding the model objects to stage vectors.

```
28------- vector<Block*> stage1;
29------- stage1.push_back( kinem);
30------- stage1.push_back( control1);
31------- stage1.push_back( airframe1);
32------- stage1.push_back( motion);
33------- stage1.push_back( target);
34------- stage1.push_back( output);
35-------
36------- vector<Block*> stage2;
37------- stage2.push_back( kinem);
38------- stage2.push_back( control2);
39------- stage2.push_back( airframe2);
40------- stage2.push_back( motion);
41------- stage2.push_back( target);
42------- stage2.push_back( output);
43-------
44------- vector<Block*> kv; // kill vehicle
45------- kv.push_back( kinem);
46------- kv.push_back( gimbal);
47------- kv.push_back( guide);
48------- kv.push_back( control);
49------- kv.push_back( airframe);
50------- kv.push_back( motion);
51------- kv.push_back( target);
52------- kv.push_back( output);
```

Within a stage, the model objects will be executed in the order that they are "pushed" onto the stage.

The train-of-objects is completed by adding these stages to a missile vector:

```
53------- vector< vector<Block*> > missile;
54------- missile.push_back( stage1);
55------- missile.push_back( stage2);
56------- missile.push_back( kv);
```

Note the self-similar (fractal-like) data structure organization used to, first, build a stage and then the same structure is used to build the missile.

> Special Note:
> Due to its clarity, this code itself can be used for documentation purposes. It's a "live-code" architecture specification. The code organization lends itself to easy interpretation, even for a system as complex as this. Clearly, this architecture is very scalable as a systematic way to organize systems with complex topologies.

Whether it is a simple system, like in the previous examples, or complex, the building process always culminates in constructing an object from the *Sim* class[58] and executing its *run* method.

```
57------- double dts[] = { dt, dt, dt};
58------- Sim *sim = new Sim( dts, tmax, missile);
59------- sim->run();
```

Next, let's look inside some of the models to see how OSK features are utilized to implement
some of the behaviors required as the stages execute.

5.4 Putting OSK to Use in the Models

The 6DOF simulation models are a good place to illustrate how various aspects of the OSK can
be utilized to implement specific functional features. As shown by these examples, these
features can be used to model a wide variety of specialized behaviors.

5.4.1 Pure Continuous Physics Model

Recall in Section 4.2, we distinguished between discrete and continuous models. The 6DOF
simulation is a good place to illustrate practical variations of these types of models.

The physics models, representing the missile's and target's dynamics for instance, are purely
continuous since all of their state derivatives evolve continuously with time. These are
frequently called the "truth" models.

The target model illustrates that a pure continuous model has no *sample* functions; instead all the
derivatives are simply defined in the model's *update* method.

```
60-------void RVTraj::update() {
61-------  // Altitude varying ballistic coefficient model
62-------  atm->update( ht);
63-------  double rho = atm->rho_();
64-------  double vrv = sqrt( vxt * vxt + vyt * vyt + vzt * vzt);
65-------  double beta = beta_vs_alt->interp( ht);
66-------  double tmp = -0.5 * G * rho * vrv / beta;
67-------  double re = RE / ( RE + ht);
68-------  vdxt = tmp * vxt;
69-------  vdyt = tmp * vyt;
70-------  vdzt = tmp * vzt + G * re * re;
71-------  ht = -zt;
72-------  p_rv( xt, yt, zt);
73-------  v_rv( vxt, vyt, vzt);
74-------}
```

5.4.2 Dedicated Discrete Avionics Model

The 6DOF simulation provides ample opportunity for implementing pure discrete models. The
flight software executing on the missile's avionics computers is a sampled data process, updating
at a particular rate to issue commands to the missile's hardware. An example of this is the
guidance law where proportional navigation is used to compute periodic acceleration steering
command updates. Here, the entire *update* method is wrapped in an *if* block[76-85] with the *sample*
function to discretize the calculations. The update rate is the argument passed to *sample*[76].

```
75-------void Guide::update() {
76-------  if( State::sample( sdt)) {
77-------    // Proportional navigation using inertial line-of-sight rates
78-------    double ad = gimbal->psied_();
79-------    double ed = gimbal->psiesd_();
80-------    double aycmd0 = vr * pngain * ad;
81-------    double azcmd0 = vr * pngain * -ed;
82-------    double accmax = cnmax * s * q_est / amass;
83-------    aycmd = limit( aycmd0, accmax);
84-------    azcmd = limit( azcmd0, accmax);
85-------  }
86-------}
```

5.4.3 Sampled Data Measurements – Hybrid Model

The real power of the *sample* function is the ability to seamlessly mix or embed discrete processes within continuous dynamics to build a hybrid model. In the 6DOF simulation, the gimbal model is an example of a hybrid model. Here, discrete target Line-Of-Sight (LOS) measurements are used to drive a continuous servo that attempts to point the seeker at the target. The seeker update process is discrete while the gimbal pointing dynamics are continuous.

The gimbal model illustrates how easy it is to model these mixed processes. The seeker model uses the relative range vector[88], *rtmb*, to calculate the seeker pointing errors, *ep* and *ey*[91]. These pointing errors are subsequently used to update the two-axis gimbal state derivatives[94,95].

```
87-------void Gimbal::update() {
88-------  rtmb = kinem->rtmb_();
89-------  ...
90-------  if( State::sample( sdt)) {
91-------    seeker( ep, ey);
92-------  }
93-------  ...
94-------  edd = akil * ( ep * akol - psiesd);
95-------  add = akil * ( ey * akol - psied);
```

This code fragment clearly illustrates the importance of properly synchronizing a model's calculations. The sequence of the calculations, updating the range vector[88], calling the seeker[91], and defining the servo state derivatives[94,95] must be performed in the order shown. Otherwise, a variable will be updated using "stale" values and the model will have a lag that is an artifact of the simulation and not truly representative of the process being modeled.

Special Note:
This example also illustrates why the architectural decision in designing the OSK was *not* to decompose a model's continuous and discrete updates into two separate methods called by the kernel. Clearly that would not have been possible for this model while maintaining proper synchronization.

It becomes a chicken-or-the-egg dilemma. If the discrete calculations are performed first[91], the seeker measurement is using a range vector, *rtmb*[88], that has not yet been defined for the current time. On the other hand, if the continuous calculations[88],[94,95] are performed first, the pointing errors[91] in the servo derivatives are not yet defined. The only recourse is to break the gimbal model into two separate models so that the combined execution of their separate discrete and continuous update methods results in the correct synchronization. This adds considerable complexity to the process being modeled (solely due to the simulation architecture) and makes it difficult to observe that proper synchronization will be maintained if the models are changed.

5.4.4 Triggering Staging

Of course, the train-of-objects architecture is ideally suited for simulating rockets or missiles with multiple stages. Within a stage model, transition to the next stage is easily accomplished using the *Sim::stop*[99] class variable as described earlier in Section 4.5.4.

```
96-------void Control_boost::update() {
97-------  if( State::sample( sdt)) {
98-------    if( State::t + EPS > t_stage) {
99-------      Sim::stop++;
100-----    }
101-----  }
102-----}
```

A stage separation is signaled when the value of this variable changes. The *Control_boost* class is used to create *both* the first and second stage objects so simply changing *Sim::stop* from its initial value of zero to a new constant will not work – the first stage separation will occur but no subsequent stage separation signals will be sent (since *Sim::stop* never changes thereafter). Therefore, *Sim::stop* is simply incremented when it is time to stage.

> Special Note:
> This usage of the kernel's *Sim::stop* class variable illustrates a key objective behind the kernel's API's: keep the APIs small (so they are easy to remember) but build as much functionality into them as possible.

5.4.5 Changing Model Connections "On-the-Fly"

Staging within the 6DOF simulation poses a unique functional requirement for the equations-of-motion class: a common *motion* object is used for all three stages but the access functions for its input change with each stage. Thus, the access functions that *motion* uses must be changed as the simulation executes.

Conceptually, this is solved by letting the data source object tell *motion* that it will be supplying the access function. In this case, the *motion* object requires forces from each stage's *airframe* object.

This is programmed as follows. First, a pointer to the *motion* object is made available to all the *airframe* objects in the main program. This will give each *airframe* the ability to execute any public methods in the *motion* class.

```
103 ----- airframe1->getsFrom( control1, motion, kinem);
104 ----- airframe2->getsFrom( control2, motion, kinem);
105 ----- airframe->getsFrom( control, motion, kinem);
```

Then we simply specify the *motion* object's data source objects at the beginning of each stage in
its *init* method.

```
106 ----- void Airframe_boost::init() {
107 -----   if( initCount == 0) {
108 -----     motion->getsFrom( this, kinem);
109 -----     t_stage0 = State::t;
```

Within each *airframe* object, the *getsFrom* method in motion is re-executed at the beginning of
each stage and an instance of the *airframe* object itself [108], *this*, is passed to it. The *motion* object
subsequently uses the correct *airframe* object's access functions to gather input.

5.4.6 Terminating the Simulation

The termination time of the 6DOF simulation is not known a priori. It is desired to conclude the
simulation when the missile "passes" the target (hopefully hitting it first). Miss distance and the
time-to-go to pass the target is tabulated in the *Kinem* class. As shown in **Figure 22**, the
kinematics class is a truth model that keeps up with the relative states between the interceptor
missile and the target. Miss-distance and time-to-go are tabulated at each time-step as part of
this process. The time-to-go variable changes sign when the missile passes the target and the
simulation should terminate.

This is implemented in code as:

```
110 ----- void Kinem::update() {
111 -----   ...
112 -----   amiss = miss( rtmb, vtmb, tgo);
113 -----   ...
114 -----   if( tgo >= 0.0) {
115 -----     Sim::stop = -1;
116 -----   }
```

First, the miss-distance is tabulated [112]. Then, if *tgo* changes sign and becomes positive, the
Sim::stop class variable is used to stop the simulation. Recall, setting *Sim::stop* to any value less
than zero signals the kernel to terminate execution.

Special Note:
We still have to specify a "hard" termination time, *tmax*, when we create the simulation from the
Sim class in the main program. The simulation will always stop at this time if nothing else
happens to terminate execution. It is always a good idea to set the *Sim* class termination time
parameter to a reasonable maximum-expected time to guarantee that the simulation will
terminate if, for some reason, "stop" signals in the simulation fail to execute due to a bug or
unexpected circumstance.

---

6   SUMMARY

The fundamental CMD design objective is to provide a concise set of C++ APIs to build basic simulation applications quickly. At the same time, CMD provides a rich set of extensible functionality underneath an easy-to-use veneer for addressing complex simulation construction requirements. Thus, a beginner can rapidly construct simple simulations in C++ with a tool that will gracefully continue to satisfy more complex modeling requirements as the user's experience with the tool grows.

The examples presented in this User's Guide have reflected the anticipated adoption and usage of CMD for a new user. Successive examples, each building on each other to illustrate more features, are designed to be worked through sequentially. A final example, a complete working 6DOF simulation, demonstrates that the OSK train-of-objects architecture paradigm scales well. As evident from the comprehensiveness of the examples, a great deal of thought has been given to what core set of capabilities are needed for effective simulation application. The user should now be well-equipped to build productive simulations with reusable models.

What is the path forward for CMD development itself? There is no path forward. As the word "kernel" implies, CMD's OSK is meant to be the simulation-equivalent of a Unix tool – additional features will be implemented not by a perpetual process of adding code to the CMD kernel itself but, instead, using the kernel in concert with other tools tailored for a particular application. A Digital Glue approach [14] using scripting languages is currently being used very successfully to configure and orchestrate automated analyses with CMD. Of course, no software is perfect and the *existing* capabilities of the CMD open-source kernel will be improved as the opportunity presents itself.

The appendixes contain a wealth of information that many CMD users will want to have. Appendix 1 describes several utility classes, besides the CMD kernel, that would benefit any simulation development. Appendix 2 describes how to use C++'s O-O inheritance feature to add additional numerical integration algorithms. Even users that have no interest in adding their own will want to look at this section to see how to use the additional integrators that have already been built and are included in the distribution (only one statement has to be added in the main program to do this). As mentioned earlier, top-level descriptions of the 6DOF simulation models are included for those that want to investigate this further. Finally, a very readable "Informal Requirements Specification" is included that gives additional insight into the design philosophy behind CMD.

While a number of high-quality, commercial tools exist to model dynamic systems (missiles included), there are advantages to building simulations at the source code level for reasons of economics, reuse of existing code, runtime issues, computer constraints, licensing restrictions, and several others. What is missing (commercial or otherwise) is a readily available tool to build simulations starting at the source-code level. CMD attempts to fill this niche.

References

1.   Danby, J.M.A. *Computer Modeling: From Sports to Spaceflight…From Order to Chaos*, Willman-Bell, Richmond, VA, (1998).

2.   Eckel, Bruce. Thinking in C++: Introduction to Standard C++, Volume One, Prentice-Hall, (2000).

3.   Stroustrup, Bjarne. The C++ Programming Language, Addison-Wesley Pub. Co., (1997).

4.   Crenshaw, J.W. Math Toolkit for Real-Time Programming, CMP Books, Lawrence, Kansas, (2000).

5.   McCarter, J., Krupp, D., and Dawson, T., Marshall Aerospace Vehicle Representation in C (MAVERIC), www.nasatech.com/briefs/apr00mfs31338.html.

6.   Sells, H.R., "Anti-Satellite Digital Testbed," 18th Space Control Conference, MIT Lincoln Laboratory, 13 April 2000.

7.   Sells, H.R. and Cooper, L. "Anti-Satellite Digital Testbed – Lessons-Learned During GN&C/Seeker Model Integration," briefing to Boeing, Anaheim, CA, 8 June 1999.

8.   Sells, H.R., Edgemon, D., Salter, A., and Schrenk, M. "Toward Establishing an Architectural Standard for Dynamical System Simulations," 2001 Summer Computer Simulation Conference, Society for Computer Simulation, Orlando, Florida, July 15-19, 2001.

9.   Sells, H.R. "Using a Scripting Language to Construct an Object-Oriented Simulation Environment," 2002 Huntsville Simulation Conference, Society for Computer Simulation, Huntsville, AL, October 9-10, 2002.

10. Sells, H.R. "TFrames User's Guide," Teledyne Brown Engineering for U.S. Army Space and Strategic Defense Command, Huntsville, AL, 7 September 1995.

11. McKerley, C.W. "Lecture Notes – Missile Guidance and Control," Nichols Research Corporation, short course notes for Southeastern Research Institute.

12. Brewer, Van, "A Missile Simulation Primer (Lecture Notes & Code Listings," Nichols Research Corporation for U.S. Army RDEC, Redstone Arsenal, AL, 1991.

13. Sells, H.R. "TFrames Missile Library Guide," Teledyne Brown Engineering for U.S. Army Space and Strategic Defense Command, Huntsville, AL, 7 September 1995.

14. Sells, H.R., "Scripted Architecture: An Alternative Way to Build Defensive Missile System Simulations," 2000 Summer Simulation Conference, Society for Computer Simulation, Vancouver, British Columbia, July 16-20, 2000.

15. Sells, H.R. "Missile Six Degree-of-Freedom Simulation Development: A User-Focused Approach," Summer Computer Simulation Conference, Society for Computer Simulation, 1995, Ottawa, Ontario.

Acronyms

6DOF	Six Degree-Of-Freedom
API	Application Programming Interface
CKEM	Compact Kinetic Energy Missile
CMD	C++ Model Developer
DE	Differential Equation
O-O	Object-Oriented
OSK	Object-Oriented Simulation Kernel
RK	Runge-Kutta

Appendix 1   Utilities

A "Swiss-army-knife" of supporting utilities is almost as important as the simulation executive itself in quickly building reliable simulations. These utilities include table look-ups, vector and matrix manipulation, and data file parsers. While a plethora of excellent C++ code already exists for these functions, CMD provides classes with APIs for these functions that are consistent with each other, intuitive, and easy-to-extend. These APIs are extensively used in the 6DOF simulation example.

The design process behind these utilities was to first postulate what the most intuitive syntax would be for a function and then use the power of C++ to implement that interface, abstracting away, as much as possible, the underlying details and complexities of operation. Also, it was recognized that users will probably want to extend these classes to tailor them for their purposes. So another design goal was to make the underlying code as readable as possible for even a novice C++ user (particularly in the vector and matrix classes). Thus, the use of templates was avoided since they place a significant syntax burden on the code's readability. The vast majority of users will simply always use the floating point type *double* since this is the format of most computers' native internal computations.

> Special Note:
> There is no requirement to use these particular utility classes in CMD and others may be used in their place or with them, if desired.

The utilities have also been built inside a C++ namespace to not collide with other libraries that you may be using.

Following are descriptions of the utility classes including their usage and syntax. The classes' API syntax is so readable that actual code, cross-referenced to a narrative, is the presentation method. This format can also easily be consulted as a reference. Full working examples for the classes, that exercise all their features, are included in the distribution.

1.1 Table Look-up

Table1, *Table2*, and *Table3* classes are provided for one-, two-, and three-dimensional table operations, respectively.

Within the narrative that follows, the term *Table* class refers to any of the Table classes.

1.1.1 Using the APIs

Files for API definitions must be included at the start of the program.

```
0 --------#include "table1.h"
1 --------#include "table2.h"
2 --------#include "table3.h"
```

All the Table class APIs are in a namespace to avoid naming collisions with other utilities that might have the same function name. All appearances of *Table1*, *Table2*, and *Table3* in the code must be prefixed by *tframes::* if the namespace is not specified.

3 --------using namespace tframes;

1.1.1.1   One-Dimensional Table

A *Table1* class[5] is provided for creating one-dimensional tables. Here we create a *tab1* pointer object that is dynamically allocated off of the heap with the *new* function[5]. *fname* specifies the data file where the tabular data resides (the format of the data file will be discussed next).

4 -------- char *fname = "table.dat";
5 -------- Table1 *tab1 = new Table1(fname);

If the *namespace* specifier is not present, the following syntax must be used to create the *tab1* object

6 -------- tframes::Table1 *tab1 = new tframes::Table1(fname);

A *read* method[7] loads the table data from the file. The tag *thrust profile* specifies where in the data file to start reading the table. *true* indicates that the table will be echoed out to the standard output device (i.e. the console).

7 -------- tab1->read("thrust_profile", true);

Now we are ready to use the table. The function *interp* performs a one-dimensional linear interpolation. Last values are returned if the bounds of the table are exceeded. This behavior is identical for all Table objects, including two- and three-dimensional tables.

8 -------- y = tab1->interp(1.412);

There is another way to define a table object. Here *t1* is on the local stack memory (as opposed to being allocated off the heap like *tab1).

9 -------- Table1 t1(fname);
10------- t1.read("thrust_profile", true);
11------- y = t1(1.304);

1.1.1.2   Two-Dimensional Table

Two-dimensional tables are used in a similar manner using a *Table2* class.

12------- Table2 *tab2 = new Table2(fname);
13------- tab2->read("cd_table", true);
14------- y = tab2->interp(2.0, 11.0);

A C++ reference variable can also be assigned to a class pointer object.

15------- Table2 &t2 = *tab2;
16------- y = t2(4.0, 11.0);

1.1.1.3   Three-Dimensional Table

A three-dimensional table class, *Table3*, is also available.

```
17-------  Table3 *tab3 = new Table3( fname);
18-------  tab3->read( "cd_table_thrust", true);
19-------  y = tab3->interp( 1.5, 8, 5);
20-------  Table3 t3( fname)
21-------  t3.read( "cd_table_thrust", true);
22-------  y = t3( 1.6, 8, 5);
```

The C++ << operator is overloaded for inclusion of tables in streams (in this case directed to the console).

```
23-------  cout << *tab1 << endl;
24-------  cout << *tab2 << endl;
25-------  cout << t3 << endl;
```

1.1.1.4   Data File Representation

The tabular data itself can reside anywhere in any ASCII text data file. The data file is specified when the table object is declared. Within the file, the tables are represented in a very readable format as they might appear on a printed page.

1.1.1.5   One-Dimensional Table

A one-dimensional table is specified as

```
26-------thrust_profile
27-------5
28-------time thrust
29-------0. 0.
30-------1. 1000.
31-------2. 2000.
32-------3. 3000.
33-------4. 4000.
```

thrust_profile[26] is the name of the table and is used to locate the table data within a file. The identifier is passed in the Table class *read*[7,10,13,18,21] method. The data can reside anywhere in the file and the table data does not have to appear in the order that the tables are read. The only restriction is that the data must appear in the order shown following the tag[26]. The next entry[27] is the number of data points. As seen, the data is represented in rows where each row is a data point – the independent value followed by the dependent value. Labels identifying the data [28] are specified prior to the numeric values themselves. There must be two space-delimited labels; therefore each label must be a single word (an underscore can be used to concatenate multi-word labels, if desired). The labels are present for readability only and are not used by the Table classes.

The numeric values appear as pairs[29-33], one set of data per line. Within a line, the format is free-form in that one-or-more space characters are used as the delimiter.

1.1.1.6   Two-Dimensional Table

A two-dimensional table is specified as

```
34-------cd_table
35-------4 3
36-------mach
37-------1. 2. 3. 4.
38-------alph
39-------5. 10. 15.0
40-------cd
41-------5. 10. 15.
42-------10. 20. 30.
43-------15. 30. 45.
44-------20. 40. 60.
```

There is some similarity to the one-dimensional table. The table data begins with a tag [34] that was specified in the class *read* function [13]. Following is the size of the table [35]. This time, because the table has two sets of independent variables, these two sets must be specified separately before the dependent data. A label for the first independent variable is declared [36] followed by the values [37]. For large tables, the independent variables can span many lines since they are parsed by ignoring "white-space" characters, including the carriage return. The parser simply keeps reading in values until the number specified [35] is reached. Labels and values are similarly declared for the second independent variable [38,39].

Finally, the dependent data itself appears in row order[41-44] preceded by a label[40] (this label is for informational purposes only and not used by the Table class). The rows correspond to the first independent variable and columns span the second independent variable. Again, rows of data may span multiple lines for large data sets. The important thing to remember is that it is read in row order where all the columns for a row are read before starting to read the following row.

1.1.1.7   Three-Dimensional Table

A three-dimensional table is specified as

```
45-------cd_table_thrust
46-------4 3 2
47-------mach
48-------1. 2. 3. 4.
49-------alph
50-------5. 10. 15.0
51-------thrust
52-------1. 10.
53-------cd
54-------5. 10. 15.
55-------10. 20. 30.
56-------15. 30. 45.
57-------20. 40. 60.
58-------
59-------50. 100. 150.
60-------100. 200. 300.
```

61-------150. 300. 450.
62-------200. 400. 600.

From the previous descriptions, it should be pretty easy to extrapolate how the data is represented. The size of the table is specified [46] followed by specification of the three independent variables [47-52].

For clarity, the independent data itself is organized into "blocks" – each block [54-57] [59-62] corresponds to successive values of the final independent variable [52]. Within a block, the data appears in row order, similar to the two-dimensional table. Cross-referencing the table size parameters [46] to the data entries themselves [54-62] should make this clear. The blank line is purely a visual aid to enhance the readability of the data and is not necessary.

1.2 Vector/Matrix/Quaternion

Tuples, and their operations, are the "nuts-and-bolts" of many math models. The vast capabilities of C++ can be harnessed to make a quite intuitive API for their use. The most exciting capability is provided by C++ operator overloading so that tuples can be manipulated using normal mathematical operators as they might appear in written text. Overloading the C++ insertion operator, <<, makes it easy to output the full contents of the tuples to streams. Finally, C++'s O-O foundation provides ample opportunity to encapsulate tuple storage and manipulation into concise and consistent modular APIs.

This section documents three tuple classes that have been tailored for coordinate system representation and manipulations in three-dimensional space. These tuple classes provide capabilities for three-element vectors, three-by-three matrices, and quaternions. These classes are particularly applicable for modeling motion in a three-dimensional coordinate space, including rigid body (translation and rotation) dynamics, but are applicable anywhere similar computations are needed.

Although they provide sufficient functionality for the example 6DOF simulation, the functionality of these classes is certainly far from complete. Their functionality is easily extended by the C++ inheritance mechanism. They were designed to serve as base classes from which to derive classes with additional methods to extend their capability.

Example programs that exercise all three classes are included in the distribution in their respective directories.

1.2.1 Vec Class

The *Vec* class provides for storing and manipulating three-element tuples.

The file for API definitions must be included at the start of the program.

0 --------#include "vec.h"

All the *Vec* class APIs are in a namespace to avoid naming collisions with other utilities that might have the same name. All appearances of the *Vec* class specifier in the code must be prefixed by *tframes::* if the namespace is not specified.

1 --------using namespace tframes;

1.2.1.1 Creation

Vectors can be declared with two kinds of constructor calls. The vector can be initialized at creation[3] or just simply declared[4]. All the elements of a vector are automatically initialized to 0 if not they are not initialized.

2 --------Vec v0, v1(10., 20., 30.), v2, v3, v4;
3 --------Vec v5(-1.0, -2.0, -3.0);
4 --------Vec v6, v7, v8;

Once a vector is created, it contents can be easily added to streams.

5 --------cout << v1 << endl;

1.2.1.2 Accessing and Assigning

There are two alternate ways to access and assign vector contents. Numerical indexing[9-11] is useful if an external counter is used to reference vector elements inside a control loop.

6 --------v2.x = -100.0;
7 --------v2.y = -200.0;
8 --------v2.z = -300.0;

9 --------v2[0] = 100.0;
10-------v2[1] = 200.0;
11-------v2[2] = 300.0;

Special Note:
The vector class also has a magnitude instance variable for storing a vector's magnitude. This is discussed later.

An *extract* method[13] provides an easy way to extract the contents of a vector into separate scalars.

12-------double phi, theta, psi;
13-------v2.extract(phi, theta, psi);

Once a vector is created, its elements can be reassigned.

14-------v3(-1.0, 2.0, 3.0);

The reassignment can take place in conjunction with other operations. Here, the contents of *v3* are changed and then *v4* is loaded with contents in *v3*.

15-------v4 = v3(-100.0, -200.0, -300.0);

There is a shorthand notation for loading the contents of a vector with the same value for each element.

```
16-------v6 = 0.0;
17-------v7 = 1.0;
```

1.2.1.3   Class Design Note

One important point regarding the vector class design needs to be made before describing vector operations. All vector class methods return their results *explicitly* like a normal function such as

```
18-------v1 = v1.scale( 10.0);
```

Here the results of the operation scaling the contents of *v1* are explicitly returned and then used to reassign *v1.*

The other design alternative would be to have some class methods operate on the vector contents *in-place* such as

```
19-------v1.scale( 10.0);
```

This has two problems. First, sometimes it may desirable for some methods to change the vector's contents in-place and sometimes not. This places an additional burden on using the vector APIs because now it must be remembered which way a particular function acts (and it is not obvious in the API's syntax itself).   Second, and sometimes a consequence of the first problem, a vector may be mistakenly used without recognizing that its contents have been changed. Changing the contents in-place tends to "mask" that the vector has been changed.

Therefore, the hard-and-fast rule is no vector methods operate on their contents in-place but instead return their results. This is also true for the *Mat* class which is described in the next section.

1.2.1.4   Operations

The addition and subtraction operations are overloaded.

```
20-------v3 = v1 + v2;
21-------v4 = v2 - v1;
```

The multiplication and division operators are overloaded for scaling the contents of a vector by a scalar. Here the contents of *v4* are scaled by 10 and the contents of *v3* scaled by 0.1.

```
22-------v3 = v4 * 10.0;
23-------v3 = v3 / 10.0;
```

Alternative ways to scale a vector are

```
24-------v3 = v1.scale( 0.1);
25-------v3 *= 0.1;
```

Consistent with the architecture that no vector operations are performed in-place, the contents of *v1* do not change when the *scale* method is applied.

The *Vec* class has an instance variable for storing the magnitude of a vector.

26-------v3.m = sqrt(v3.x * v3.x + v3.y * v3.y + v3.z * v3.z);

This manual computation is not necessary since a *mag* function is provided. Note that the *mag* function does not set the *.m* magnitude variable. This must be done explicitly with an = assignment.

27-------v3.m = v3.mag();

```
Special Note:
The magnitude component of a vector must always be calculated explicitly if it is needed. The
alternative would have been to have the magnitude automatically updated every time the vector's
contents change.  This would introduce an additional computational burden to all vector
operations which may not be needed in many cases.
```

Having the *mag* function set the vector *.m* magnitude in-place would also violate the design criteria that no vector methods update vector contents in-place.

Unit vectors are created using the *unit* class method. Again, an explicit unit vector is returned and the contents of *v3* are not affected.

28-------v4 = v3.unit()

Finally, an advanced capability is provided that is related to those in functional programming languages. A function can be applied to all the elements of a vector simultaneously

29-------v3 = v5.apply(fabs);

Here we used the standard C++ math *fabs* library function. You can also supply functions of your own:

30-------v3 = v5.apply(add1);

where you define *add1* as a normal function:

31-------double add1(double x) {
32------- return x + 1.0;
33-------}

The *Vec* class provides functions useful for coordinate transformations and manipulations. First are vector dot and cross products

v2 = v0.dot(v1);
v2 = v0.cross(v1);

Second are functions for transforming a vector of Euler angles representing a coordinate system rotation in a heading-elevation-roll sequence. A direction cosine matrix is defined from the vector of Euler angles

Mat mDCM;
mDCM = vEuler.getDCM();

where *vEuler* is a vector containing the Euler angles (rad). A three-by-three matrix is returned. The *Mat* class is described in the following section, 1.2.2. A similar function for extracting the quaternion is

Quat q0;
q0 = vEuler.getQuat();

Where q0 is the quaternion rotation represented by the three Euler angles. The *Quat* class is described in section 1.2.2.4.

The inverse methods for *getDCM* (return Euler angle vector given the direction cosine matrix) and *getQuat* (return Euler angle vector given the quaternion) are described in the *Mat* and *Quat* class descriptions, respectively.

1.2.2 Mat Class

The *Mat* class provides for storing and manipulating three-by-three matrices.

A file for the API definitions must be included at the start of the program.

0 --------#include "mat.h"

The *Mat* class is designed to be used in conjunction with the *Vec* class. All the *Vec* class API definitions are included in mat.h.

All the *Mat* class APIs are in a namespace to avoid naming collisions with other utilities that might have the same name. All appearances of the *Mat* class specifier in the code must be prefixed by *tframes::* if the namespace is not specified.

1 --------using namespace tframes;

1.2.2.1   Creating

Matrices can be declared with three kinds of constructor calls. The matrix can be initialized at creation or just simply declared. All the elements of a matrix are automatically initialized to 0 if not they are not initialized.

2 --------Mat m3(10., 20., 30., 40., 50., 60., 70., 80., 90.);
3 --------Mat m4(v1, v2, v3);
4 --------Mat m5;

The matrix can be initialized with the elements listed in row-order in the constructor call[2]. For example, *m3* is the matrix

$$m3 = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}$$

Vectors can also be used to initialize the matrix[3]. Consistent with the row-ordering, each vector is a row in the matrix.

Once a matrix is created, it contents can be easily added to streams.

5 --------cout << m3 << endl;

1.2.2.2 Accessing and Assigning

Matrix elements are accessed using subscript indexes.

6 --------double a;
7 --------a = m3[0][1];

Matrix contents can also be accessed as vectors.

8 --------vec v;
9 --------v = m3[0];

where *v* is the first row in *m3.*

The elements of a matrix can be reassigned

10-------m0(1., 2., 3., 4., 5., 6., 7., 8., 9.);
11-------m0(v1, v2, v3);

1.2.2.3 Operations

The addition and subtraction operations are overloaded.

12-------m1 = m0 + m3;
13-------m4 = m1 - m3;

The multiplication operator is overloaded for both vector and matrix multiplication

14-------vec v0, v1;
*15-------v0 = m0 * v1;*
*16-------m2 = m0 * m1;*

The multiplication and division operators are also overloaded for scaling the contents of a matrix by a scalar. Here the contents of *m4* are scaled by 10 and the contents of *m3* scaled by 0.1.

*17-------m3 = m4 * 10.0;*
18-------m3 = m3 / 10.0;

Alternative ways to scale a matrix are

```
19-------m3 = m1.scale( 0.1);
20-------m3 *= 0.1;
```

Consistent with the architecture that no matrix operations are performed in-place, the contents of *m1* do not change when the *scale* method is applied.

Methods for common matrix transpose, determinant, and inversion operations are provided

```
21-------m2 = m1.transpose();
22-------double d = m0.det();
23-------m1 = m0.inv();
```

As with a vector, a function can be applied to all the elements of a matrix simultaneously

```
24-------m3 = m5.apply( fabs);
```

Here we used the standard C++ math *fabs* library function. You can also supply functions of your own. For example to load a matrix with random numbers

```
25-------double rand_double( double) {
26-------  return ( double)rand() / RAND_MAX;
27-------}
```

```
28-------m3 = m5.apply( rand_double);
```

Two matrix methods useful for coordinate transformations and manipulations are provided

```
29-------vEuler = m1.getEuler();
30-------q0 = m1.getQuat();
```

where *m1* is the direction cosine matrix representing a heading-elevation-roll coordinate system rotation sequence.

Special Note:
The characteristic that all the *Vec* and *Mat* class APIs are functions that return explicit results (as opposed to changing themselves in-place) offers some convenient (and very readable) code shortcuts.

Suppose we want to quickly calculate

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^{-1} \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$$

```
31-------Vec v;
32-------Mat m;

33-------m = m( 1, 2, 3, 4, 5, 6, 7, 8, 9).inv() * v( 10, 20, 30);
```

where we use *v* and *m* simple as "dummy" objects to temporarily hold our values.

Similar economies of coding can be taken advantage of in coordinate transformations and manipulations.

1.2.2.4   Quat Class

The *Quat* class provides for storing and manipulating quaternions represented as four-element tuples.

$$q = \begin{bmatrix} \text{magnitude of rotation} \\ [\text{unit vector of rotation}] \end{bmatrix}$$

The first element in the tuple is the rotation angle component. The following three elements define the axis of rotation.

Files for API definitions must be included at the start of the program.

```
0 --------#include "quat.h"
```

The Quat class is designed to be used in conjunction with the *Vec* and *Mat* classes. All the *Vec* and *Mat* class API definitions are included in quat.h.

All the *Quat* class APIs are in a namespace to avoid naming collisions with other utilities that might have the same name. All appearances of the *Quat* class specifier in the code must be prefixed by *tframes::* if the namespace is not specified.

```
1 --------using namespace tframes;
```

1.2.2.5 Creating

Quaternions can be declared with two kinds of constructor calls. The quaternion can be initialized at creation or just simply declared. All the elements of a quaternion are automatically initialized to 0 if not they are not initialized.

```
2 --------Quat q0(0.962, -0.023, 0.084, 0.258);
3 --------Quat q1;
```

The scalar part of *q0* is the first element, 0.962.

Once a quaternion is created, it contents can be easily added to streams.

```
4 --------cout << q1 << endl;
```

1.2.2.6 Accessing and Assigning

Like a vector, there are two alternate ways to access and assign quaternion elements:

```
5 --------q0[0] = 0.962;
6 --------q0[1] = -0.023;
7 --------q0[2] = 0.084;
8 --------q0[3] = 0.258;

9 --------q0.s = 0.962;
10-------q0.x = -0.023;
11-------q0.y = 0.084;
12-------q0.z = 0.258;
```

Once a quaternion is created, its elements can be reassigned.

```
13-------q0 ( -0.332, 0.177, 0.907, -0.188);
```

1.2.2.7 Operations

The quaternion is normalized with

```
14-------q0.normalize();
```

A direction cosine matrix extraction function is provided that works in conjunction with *Mat* class objects (matrices)

```
15-------Mat m0 = q0.getDCM();
```

where the direction cosine matrix is defined using a heading-elevation-roll angle transformation sequence.

1.3 Input Parsing

The necessity to read data from files is found in almost any simulation. And yet, writing code for this is one of the most tedious and error-prone tasks in building a simulation.

1.3.1 Description

CMD provides a Filer class to greatly assist in the simple task of parsing and retrieving parameter values from a data file. Parameter values with a user-specified label are listed in an ASCII text file. The parameters, values and labels, can be grouped by tags in the data file for selective parsing. Formatting within the data file is free-form so there are no restrictions on columnar positioning of the data. The data items can even be appended by explanatory comments if desired. The Filer class provides a simple API to first locate the tag (if desired) and then parse for a particular variable.

1.3.2 Data File Representation

Data is specified in a data file by variable name and then its value. White space (tabs and blank spaces) and the equal sign are the delimiters and, hence, ignored during parsing. The parser attempts to read the first two tokens on a line. If the first token matches the variable name that is being read, the second token is read as the value. Optionally, a tag may be specified to skip-to before parsing for the variable being read.

An excerpt from an example input data file is

```
0 --------xka = -44.486
1 --------xkg = 3282.29
2 --------xcp = 2.45
3 --------xcg = 1.198
4 --------
5 --------The following data came from Fred B.
6 --------Airframe
7 --------xcp = 1.448  center-of-pressure
8 --------xcg = 1.422  temporary value – needs to be updated
9 --------dia = 0.886315
10-------xji = 2.18440
11-------cmq = 0.0001
12-------cl = 0.0
```

Here the variables are specified in a user-friendly syntax. The equals signs are ignored by the parser but can be included for readability, if desired. Note that explanatory comments[7,8] can also appear after the variables are specified since the parser will only attempt to read the first two tokens. Blank lines[4] and even lines of arbitrary text[5] can be included (as long as you are careful that the first word in any extra text is not a variable name or tag!).

A very important feature is that data can be tagged with a label to instruct the parser not to begin scanning for variable input until the tag has been read. Here the tag *Airframe*[6] is used so that the intended *xcp*[7] value is read and not the one[2] that appears before the tag. This feature is useful for delimiting blocks of data in a single data file for multiple stages on a missile, for example. The stages might all use the same parameters, just with different values for each stage. The tags could be used to put the data for all the stages in one file even though the same parameter names are used on all the stages.

1.3.3 Code Implementation

The file for API definitions must be included at the start of the program.

13-------#include "filer.h"

Referencing the data file excerpt in the previous section, suppose it is desired to read the variables *xcp*, *xcg*, and *dia*[14] that are associated with an *Airframe* block of data in the missile.txt data file. The *Filer* class is used as follows

14-------double xcp, xcg, diameter;
15-------Filer *ff = new Filer("missile.txt");
16-------ff->setLine0("Airframe");
17-------xcp = ff->getDouble("xcp");
18-------xcg = ff->getDouble("xcg");
19-------diameter = ff->getDouble("dia");

A *Filer* object *ff*[15] is first created passing the name of the data file to the constructor so the parser knows where to look. The *setLine0* method[16] is used to specify to start trying to read the variables only after the tag *Airframe* has been found. Parsing will begin at the first line in the data file if the *setLine0* method is not used. Here, it is important to use the tag because the variable names appear multiple times[2,7][3,8] in the data file and we need to specify which ones are to be read. The variables themselves are loaded using the *Filer* class *getDouble* function[17-19]. The name of the label designating the parameter to be read is passed to the function. The label of the parameter in the data file does not have to necessarily match the variable name. An error message is returned if the parameter name is not found.

Two additional APIs are provided to read integer and string type parameters

20-------int n_targets = ff->getInt("n_targets");
21-------string run_id = ff->getString("run_title");

The rules for using the *Filer* class parser, while simple, provide a powerful and expedient means to load data into the simulation and can save the simulation builder a lot of work. This is the data input method used in the 6DOF missile simulation example.

Appendix 2   Adding an Integration Algorithm

One feature claim that CMD *does not* make is that it has the best numerical integration algorithm (although the standard fourth-order Runge-Kutta (RK) algorithm implemented in the kernel's *State* class does a pretty good job). Perhaps it's best to "table" this issue with the observation that the best algorithm is left up to the analyst to choose based on the system that is being simulated.

With this in mind, it becomes an important simulation feature to allow the developer to implement a new algorithm, if desired. The CMD kernel was designed to be extensible so that new integrators can be easily added while, at the same time, not changing any of the existing code in the kernel. The C++ inheritance feature was leveraged to provide this capability.

We'll illustrate the process by implementing a second-order RK numerical integration algorithm and then using it. This topic concludes with a discussion of an additional algorithm that has been implemented and is highly recommended. Remember, if you don't like these – CMD let's you "Roll your own!"

Special Note:
This section will be of interest even if you are not contemplating building your own algorithm. The distribution includes ready-to-use algorithms for Euler's method, 2nd Order RK, and Merson's method. Consult section 2.4 on how to use these instead of the default 4th Order RK method.

The completed code for the additional integration algorithms is in directory ex_7 of the distribution.

2.1 Second-Order Runge-Kutta Algorithm

A 2nd Order RK method was selected for illustration because it is complex enough to fully illustrate the mechanics of adding a custom algorithm while, at the same time, simple enough not to obfuscate the mechanics itself.

The formula for a second-order RK numerical integration algorithm is

$$y_{j+1} = y_j + \Delta t\, f(y^*_{j+1/2}, t_{j+1/2})$$

$$y^*_{j+1/2} = y_j + \frac{\Delta t}{2} f(y_j, t_j)$$

$$t_{j+1/2} = t_j + \frac{\Delta t}{2}$$

Note that this is a *multiple evaluation per time-step* method in that the derivatives are updated multiple times within an integrating time-step. This is a characteristic of many of the more accurate methods and thus was a characteristic selected for this example.

2.2 Inheriting from the Kernel

New integrators are built by inheriting from the existing *State* class in the kernel. This way we preserve use of much of the kernel's functionality, which can be quite complex, and only have to change those functional aspects that are unique to the new algorithm. The *State* class was designed so that only two of its methods need to be overridden for a new integrator – one that actually propagates the states and one that controls the clock. All the other kernel functionality (operation of the *sample* function, for instance) is automatically provided by existing methods that are inherited and need not concern the coder.

The process begins by declaring a new class:

```
0 --------class State_rk2 : public State {
1 --------  public:
2 --------    State_rk2() : State() {}
3 --------    State_rk2( double &x, double &xd) : State( x, xd) {}
4 --------    State_rk2 *factory( double &x, double &xd) {
5 --------      return new State_rk2( x, xd);
6 --------    }
7 --------    void propagate();
8 --------    void updateclock();
9 --------  protected:
10-------};
```

A class is declared that derives from the kernel's *State* class[0]. A constructor must be specified that passes the state and state derivative to the *State* base class[3]. The OSK will need the capability to create new state objects as needed so a *factory* method is declared[4] that simply creates a new instance of itself. This will always be the same, regardless of the algorithm. This method is used internally by the OSK to create states when an integrator is added. The last thing to do is to declare new *propagate*[7] and *updateclock*[8] methods. These are the two methods that need to be overridden to propagate the states and control the clock and are the things to be changed to implement a new algorithm.

Next, let's examine how *propagate* and *updateclock* are built.

2.3 Coding the New Algorithm

The *propagate* method is coded directly from the algorithm, so let's look at it first.

The new *propagate* method is

```
11-------void State_rk2::propagate() {
12-------  switch( kpass) {
13-------    case 0:
14-------      x0 = *x;
15-------      xd0 = *xd;
16-------      *x = x0 + dt / 2.0 * xd0;
17-------      break;
18-------    case 1:
19-------      xd1 = *xd;
20-------      *x = x0 + dt * xd1;
```

```
21------     break;
22------  }
23------}
```

The purpose of this method is to propagate the states based on the derivatives. For multiple evaluation algorithms, the kernel uses the variable *kpass*[12] to switch between formulas within the time-step. The two *case* blocks[13][18] represent the two evaluations in the equation. Within these evaluations, the current state, *x*, and state derivative, *xd*, are accessed by pointers. These values should be saved within an evaluation, if they are required in a subsequent evaluation. The states are propagated within each block[16][20].

To control the clock, the *updateclock* method is overridden. This method is responsible for incrementing the clock and controlling the current time, *t*. To do this, you need to control four *State* class variables: *kpass*, *ready*, *t*, and *t1*. Let's discuss these parameters' functions before looking at the code.

As we have seen, *kpass* is an integral counter used to keep up with the number of evaluations within a time-step. *kpass* is zero at the time-step boundaries and should be incremented by 1 on each pass (each derivative evaluation) within the current time-step. It should be set back to zero after all the passes are completed. *kpass* is used elsewhere by the kernel (in *sample*, for instance) so it is important to increment it properly with each pass.

Related to *kpass*, the variable *ready* is used elsewhere by the kernel to signal that the simulation is at a time-step boundary. You are responsible for setting *ready* = 1 at *kpass* = 0 and setting *ready* = 0 otherwise.

The *State* class variable *t* is the simulation time. This is the same variable *State::t*, used in the models for the current simulation time. The *updateclock* method is responsible for maintaining and incrementing *t*. This includes "sub-incrementing" *t* during the multiple passes within an integrating time increment consistent with the numerical integration algorithm. Sub-incrementing *t* within a time-step is done in synchronization with *kpass* to accomplish the derivatives' evaluation at the correct time.

The final *State* class variable is *t1*. The kernel uses *t1* to specify the boundary of the next integrating time-step. You are responsible, in *updateclock*, to simply set this time to the current time plus the integration step size (*dtp* described next) at the start of an integrating time-step. On the final pass, you should also set the time, *t*, equal to *t1* to prepare for the next time-step. The kernel may adjust *t1* as the models' *update* methods are executed if a *sample* event occurs.

Two parameters, that you should not change, are available in *updateclock* to help you increment time. *dtp* is the default integrating time increment that was specified when the stage was created in the *Sim* class object. Use *dtp* to increment *t* to *t1* at the start of the integrating time-step to establish a starting value for *t1*. However the step-size, and consequently *t1*, may change due to the occurrence of a *sample* event as the models' *update* methods are executed. This step-size (whether it changes from *dtp* or not) is reflected in variable *dt*. Therefore, the *dt* should be used to increment time, if required, within a time-step. Again, *t* should be set to *t1* on the final pass (as opposed to directly incrementing it by *dt* or some fraction thereof) to put *t* precisely on the boundary of the next time.

With this said, things should proceed as follows in *updateclock* (cross-referenced to code that follows):

1) Adjust the current time, if necessary, depending the current pass within the time-step [25-30]
2) Increment *kpass*, the pass counter [31,32]
3) If the current time is at a time-step boundary, initialize *t1* [33-38]
4) Set *ready* to reflect whether the current time is at time-step boundary (for example beginning of a new integration time-step) [33-38]

With this in mind, let's look at the *update* clock method for the 2nd Order RK.

```
24-------void State_rk2::updateclock() {
25-------  if( kpass == 0) {
26-------    t += dt / 2;
27-------  }
28-------  if( kpass == 1) {
29-------    t = t1;
30-------  }
31-------  kpass++;
32-------  kpass = kpass % 2;
33-------  if( kpass == 0) {
34-------    ready = 1;
35-------    t1 = floor( ( t + EPS) / dtp + 1) * dtp;
36-------  } else {
37-------    ready = 0;
38-------  }
39-------}
```

Time is incremented according to *kpass*[25,28]. Use *dt* to sub-increment *t* within a time-step[26]. Do not increment to the final time; instead simply set t = t1[29]. You are responsible for providing logic to properly cycle the counter *kpass*[31.32]. *ready* should be set[34,37] according to whether the current time is at a time-step boundary. If the current time is at a time-step boundary, the initial boundary for the next time-step, *t1*, should be defined[35]. You should always use the *floor* algorithm shown to precisely set *t1* to an integral multiple of *dtp* (this ensures accurate event catching in *sample*).

Overriding these two methods completes the 2nd Order RK installation. The next step is telling the kernel to use the new *State_rk2* class instead of the original *State* class.

2.4 Using the New Algorithm

Only a minor change needs to be made to the *Sim* class object to use the new integrator.

The main program to accomplish this is

```
40-------int main() {
41-------  double  tmax = 4.00;
42-------  double  dt = 0.001;
```

```
43------- // Uncomment these to declare and use new integrators
44------- //Block::integrator = new State(); //default, can declare if you want to
45------- Block::integrator = new State_rk2();
46------- Model *model = new Model( 1.0, 0.0);

47------- vector<Block*> vObj0;
48------- vObj0.push_back( model);

49------- vector< vector<Block*> > vStage;
50------- vStage.push_back( vObj0);

51------- double dts[] = { dt};
52------- Sim *sim = new Sim( dts, tmax, vStage);
53------- sim->run();
54-------}
```

The static class variable *Block::integrator*$^{<45>}$ is used to specify which numerical integration algorithm is to be used. This variable holds an instance of the integrator class. The *factory* method defined in the integrator's constructor is used to instantiate new states as needed.

We haven't had to assign *Block::integrator* before because this assignment is performed internally in the OSK as part of its initialization, but this statement can be added$^{<44>}$, if desired. That is, if no assignment is specified in the main program, the default 4th-order RK is used.

2.5 Testing the New Algorithm

Of course, a good test case is needed as you code, test, and debug a new integrator. The test case should be simple enough that it is readily coded and the results easily observed but, on the other hand, complex enough to uncover any anomalies.

Experience has proven that this DE is a good system to use for this purpose

$$\frac{d^2y}{dt^2} + ty = 0$$

$$y(0) = 1, \ \frac{dy}{dt}(0) = 0$$

with the equivalent state-space representation

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -t & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \ \ y = x_1$$

Why is this a good test case? This seemingly trivial system has some characteristics that require careful attention when modeling it in a digital simulation: the states are coupled (non-zero off-diagonal terms in the state representation) and one of them has a time-varying coefficient. These two characteristics will uncover any synchronization problems and thus thoroughly test the new algorithm.

Special Note:
One author has used this simple system to test simulations for years. It is remarkable how many "mature" simulations have failed to get the correct answer for this simple problem.

The reason many simulations fail this test is a synchronization problem. In fact, a noted simulationist has made the observation, "Getting x out of sync with t is the most common error in implementations of numerical integration algorithms, and I can't tell you how many times I've seen this mistake made," Crenshaw[4].

The simulation can be easily coded from the state-space representation. The complete model for this is

```
55-------Model::Model( double x10_, double x20_) {
56-------  x10 = x10_;
57-------  x20 = x20_;
58-------  addIntegrator( x1, x1d);
59-------  addIntegrator( x2, x2d);
60-------}

61-------void Model::init() {
62-------  cout << "starting Model...\n";
63-------  if( initCount == 0) {
64-------    x1 = x10;
65-------    x2 = x20;
66-------  }
67-------}

68-------void Model::update() {
69-------  x1d = x2;
70-------  x2d = -State::t * x1;
71-------}

72-------void Model::rpt() {
73-------  if( State::sample( 0.8) || State::ticklast) {
74-------    printf( "%8.3f %8.6f\n", State::t, this->x1);
75-------  }
76-------}
```

Running the test case yields

```
77-------starting Model...
78-------  0.000 1.000000
79-------  0.800 0.916113
80-------  1.600 0.405399
81-------  ...
82-------  4.000 0.219972
```

This is the expected answer for a properly implemented 2nd-order RK (with dt = 0.001). The exact solution is x = 0.219970 at t = 4 sec.

2.6 A Recommended Algorithm

The fully implemented 2nd Order RK is included in the distribution along with two others to further give "go-bys" to implement your own. A *State_euler* class implements perhaps the most simple (but rarely useful in practice) algorithm.

One author has found the second included algorithm to be very efficient in that it provides accurate answers with relatively large time-steps thus reducing runtime. It is called Merson's Method

$$k_1 = \Delta t \ f(x,t)$$

$$k_2 = \Delta t \ f(x + \frac{1}{3}k_1, t + \frac{1}{3}\Delta t)$$

$$k_3 = \Delta t \ f(x + \frac{1}{6}k_1 + \frac{1}{6}k_2, t + \frac{1}{3}\Delta t)$$

$$k_4 = \Delta t \ f(x + \frac{1}{8}k_1 + \frac{3}{8}k_3, t + \frac{1}{2}\Delta t)$$

$$k_5 = \Delta t \ f(x + \frac{1}{2}k_1 - \frac{3}{2}k_3 + 2k_4, t + \Delta t)$$

$$x(t + \Delta t) = \frac{1}{6}(k_1 + 4k_4 + k_5) + x(t)$$

and is found in Crenshaw [4]. He states "I was so impressed that I've been using it ever since." Note that the method requires an additional (5th) evaluation over the 4th RK but its efficiency in the ability to maintain accuracy with larger time-steps seems to more than overcome this. The formula is shown here to cross-reference to the code as an additional aide to studying how to implement your own algorithms.

The class can be used in CMD by setting *Block::integrator* equal to a new instance of the *State__mm* class in the main program

83-------Block::integrator = new State_mm();

before executing the *Sim* class *run*() method.

Appendix 3   6DOF Model Descriptions

The intent of the 6DOF simulation example was to illustrate CMD's application to model a more complex "real-world" system than the preceding tutorial examples. Also, as earlier stated, the 6DOF example was not meant to be a primer for building missile flight simulations. The constituent models were intentionally kept very basic (generally less than 100 lines of code) while having enough complexity to exercise salient CMD features in a flight simulation context. Nevertheless, it is useful to provide an introduction to the models for those users that might be interested in building similar simulations.

The following brief narratives are designed to expedite studying the code for the individual models and conform to the model categories in *Figure 22*. The title of each subsection, in parentheses, is cross-referenced to the figure. Brief descriptions are given for each model followed by a glossary of the major variables. These should be sufficient for the experienced 6DOF modeler to interpret the math models in the code.

3.1 RVTraj (Target)

This is a three-dimensional point-mass target model for the missile to attempt to intercept. The three degrees-of-freedom are translations in the inertial x, y, and z directions. The target deceleration is governed by a table of ballistic coefficients (beta tables) and the earth's (flat) gravitational field. Target drag is computed according to

$$\frac{dV}{dt} = -\frac{\frac{1}{2}\rho|V|^2}{\beta}, \beta = \frac{W}{C_d S}$$

In this model, as well as the rest of the simulation, +z axis is down; that is the altitude is –z.

Table A- 1 Target Model Major Variable Glossary

Variable	Description	Units
xt0, yt0, ht0	target initial position (inertial)	m
vxt0, vyt0, vzt0	target initial velocity (inertial)	m/sec
xt, yt, zt	target position (inertial)	m
vxt, vyt, vzt	target velocity (inertial)	m/sec
vdxt, vdyt, vdzt	target acceleration (inertial)	m/sec^2

3.2 Kinem (Kinematics)

This model calculates relative kill vehicle/target trajectory states (kinematics) including range, relative velocity, projected miss distance at point of closest approach, and time until closest approach(time-to-go). Time-to-go becomes positive when the kill vehicle passes the target and is used as the condition to terminate the simulation. Time-to-go and miss distance are calculated according to

$$t_{go} = \frac{V \cdot R}{V \cdot V}$$

$$M = R - Vt_{go}$$

where *R* and *V* are the relative range and velocity vectors, respectively.

Table A- 2 Kinematics Model Major Variable Glossary

Variable	Description	Units
Rtm	range vector, inertial	m
Vtm	relative velocity vector, inertial	m/sec
Rtmb	range vector, missile body coordinates	m
Vtmb	relative velocity vector, missile body coordinates	m/sec
g_local	gravity vector, inertial coordinates	m/sec^2
g_body	gravity vector, body coordinates	m/sec^2

3.3 Gimbal (Measure)

This is a two-axis gimbal and track loop model that is pointed at the target by a servo-loop. This class emulates the tracking function where the target motion is measured to provide information for kill vehicle guidance. The seeker platform is inertially stabilized and the rates in pitch and yaw are used as line-of-sight rates for proportional navigation in the guidance class. The gimbal has two rotational degrees-of-freedom with the outer gimbal angle moving in elevation and the inner gimbal angle moving in azimuth. The tracking servo loop consists of an outer position loop and an inner rate stabilization loop which keeps the seeker boresight axis aimed at the target. Boresight error measurements are provided by a simple seeker model which uses simple Gaussian-distributed noise to corrupt the true boresight error measurements.

Table A- 3 Gimbal Model Major Variable Glossary

Variable	Description	Units
anub, anue	outer and inner gimbal angles	rad
psiesd, psied	inertial seeker platform angular rates (y and z axes)	rad /sec
akil, akol	gimbal servo inner and outer loop gains	
wx, wy, wz	kill vehicle body rotational rates	rad /sec
ep, ey	boresight errors (pitch and yaw)	rad

3.4 Guide (Guidance)

This is a simple proportional navigation steering law. Missile steering commands in the form of lateral accelerations are generated based on missile/target line-of-sight rates provided by the tracker. The magnitude of the acceleration command is limited by a user-specified maximum product of the aerodynamic normal force coefficient and the desired maximum angle-of-attack.

Table A- 4 Guidance Model Major Variable Glossary

Variable	Description	Units
pngain	proportional navigation gain	
vr	average relative velocity	m/sec
cnmax	used to limit angle-of-attack	
s	aerodynamic reference area	m^2
amass	average mass	kg
q_est	average dynamic pressure	N/m^2
aycmd, azcmd	commanded lateral body accelerations	m/sec^2

3.5 Control (Control)

This model is a simple lateral and roll autopilot for the kill vehicle. No functions are performed in this module for the boost stages since they are ballistic. The lateral autopilot uses feedback from the accelerometers and rate gyros to compute a jet-interaction thruster command needed to achieve angle-of-attack for the desired lateral acceleration in independent pitch and yaw channels. The roll autopilot uses body rate feedback from rate gyros and gimbal angles to null the inner gimbal (azimuth) angle.

Table A- 5 Control Model Major Variable Glossary

Variable	Description	Units
xk, xka, xkg	lateral autopilot scale factor, acceleration, rate gyro gains	
thjiy_cmd, thjiz_cmd	commanded jet-interaction thrusts	N
thrl_cmd	commanded roll thrust	N
ay, az	body lateral accelerations	m/sec^2
wx, wy, wz	body rotational rates	rad/sec

3.6 Airframe (Airframe)

This is an all-in-one class to characterize the missile's physical characteristics including mass, aerodynamics and propulsion. Aero coefficients computed as a function of Mach number and angle-of-attack from two-dimensional table look-ups. Boost propulsion profiles come from one-dimensional thrust vs. time tables. The class outputs are three-component force and moment vectors.

Special Note:
Encapsulating mass, aero, and propulsion in a single class simplifies model variable "connection" problem since these three functions are usually coupled. This encapsulation provides easy means to "swap" an entire missile (i.e. airframe) by substituting this one class.

Table A- 6 Airframe Model Major Variable Glossary

Variable	Description	Units
xji	force application of jet-interaction thrust (from nose)	m
xcp, xcg	center-of-pressure, center-of-gravity (from nose)	m
xroll	roll moment arm for roll force application	m
amass	kill vehicle mass	kg
ajx, ajy, ajz	kill vehicle mass moment-of-inertias	kg m^2
alphp, alphy	pitch/yaw angles of attack	rad
force	total force vector from all forces (aero and applied)	N
moment	total moment vector from all forces	N m

3.7 Motion (Motion)

This model propagates the missile states in both translation and rotation (six degrees-of-freedom). Quaternions are used to represent attitude The input is the net force and moment vector from the Airframe class. The quaternion is normalized at the start of each time-step.

Table A- 7 Motion Model Major Variable Glossary

Variable	Description	Units
phi_deg, theta_deg, psi_deg	Computed Euler angles (yaw-pitch-roll sequence)	deg
p_eci	Missile position vector	m
v_eci	Missile velocity vector	m/sec
force	Net force input vector	N
moment	Net moment input vector	N m
quat, quat_d	quaternion and quaternion derivative	

Appendix 4   Informal Requirements Specification

As stated in the introduction, the fundamental design objective behind CMD is to provide the ability to go from mathematical model representation to efficient, working C++ code with the minimum amount of effort. This goal is somewhat difficult to capture in a formal set of requirements. What makes a simulation easy-to-use and modify? How does "easy-to-use" translate into simulation requirements?

Perhaps the starting point for addressing these questions is to ask, "How would an analyst (the one that modifies/adds to/uses the simulation on a daily basis) pose a set of requirements?" From the user's perspective, the primary roadblock to effective simulation utilization is frequently not the lack of a well organized inventory of modular physics models but instead the shortcomings and mechanization of the underlying simulation engine in which the models reside. This assertion is supported by the experience that, in many cases, the majority of simulation development time is spent on modification of the underlying simulation structure and not on the models themselves. Missile simulation development requirements frequently focus on what the simulation is to model and, to a much lesser extent, how the analyst uses and interacts with the simulation itself.

Following is a list of requirements that an analyst might pose. These requirements are based upon common pitfalls that an analyst encounters in the process of building and configuring a simulation and their solution. While these may be somewhat less specific than those requirements that might appear in a formal software design document, they provide further insight to CMD design goals. A challenge for future work is to formalize this analyst-based requirements approach so that it might augment current software engineering processes.

The requirements listed here are only a starting point but were selected because they seem to be the ones that, from a subjective viewpoint, most affect that nebulous thing called "ease-of-use." Undoubtedly, many more can be added.

This section is largely based upon previously published work, Missile Six Degree-of-Freedom Simulation Development: A User-Focused Approach [15].

4.1 Separate Model Control From the Models

4.1.1 Issue

The key to any efficient simulation development is the reuse of the constituent models. Often, however, the code for the algorithms and physics mathematical formulations (i.e. the models) cannot be cleanly segregated from the simulation control functions such as numerical integration, input/output, data handling, and order of model execution (i.e. model control). Thus if a model for a flight control algorithm or actuator model is to be reused, much of the underlying simulation control infrastructure must be carried along and made compatible with the host simulation or stripped away – both inefficient, time-consuming fixes that discourage reuse.

4.1.2 Requirement

First and foremost, the 6DOF simulation construction process should be decomposed into two steps. The missile 6DOF simulation domain can be represented from two perspectives. ***Figures A-1 and A-2*** illustrate that a missile simulation can be decomposed from a functional perspective as well as being represented, more abstractly, as a system of differential equations to be solved. Too often simulations are developed using only a functional perspective. The 6DOF missile engagement simulation is only a subset of the broader class of systems described by time-based differential equations. With this perspective, building a missile 6DOF simulation can be decomposed into a two-step process: first, building a robust engine to solve differential equations and then adding the differential equations which describe the models. A tape-player/tape analogy to this concept is illustrated in ***Figure A-3***. The advantage of this decomposition is that the underlying differential equation solving engine need only be built once and is totally reusable.
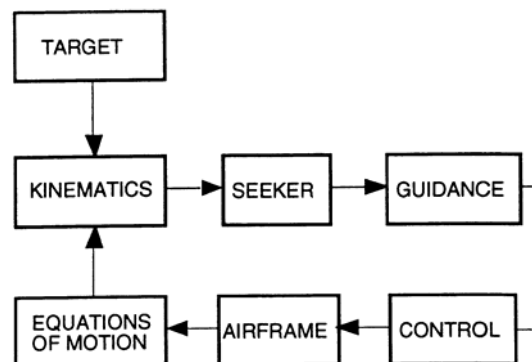

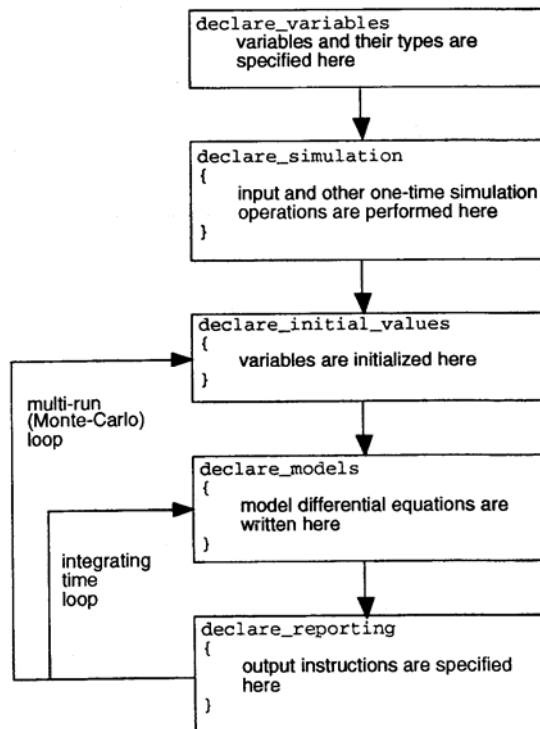*Figure A-1. Missile Simulation Functional Decomposition*

Figure A-2. Missile Simulation Differential Equation Decomposition
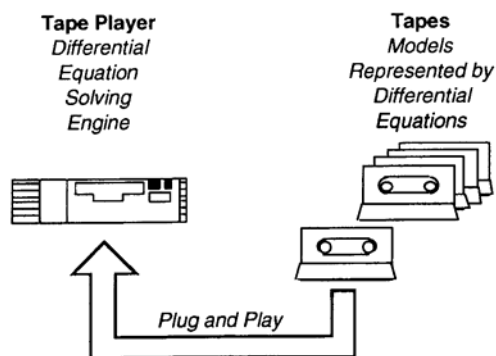

Figure A-3. Tape-Player Analogy for Simulation

The simulation should provide a robust environment for solving systems of differential equations and thus simulation development can focus on coding the equations describing the constituent missile models themselves. Typically, simulation models and code are written starting at the programming language level (Ada, C, Fortran are common examples). Thus the model developer is faced with not only coding the mathematical equations that describe the model but also defining the simulation structure that will host the model. This expects a great deal (and probably uncommon amount of) expertise on the part of the modeler in that the modeler must be a competent engineer/scientist as well as a good simulation coder. The modeler should be

provided a solid simulation structure as a starting point. A simulation is constructed by writing the mathematical equations describing the system using this structure. This structure provides the flexibility to model virtually any dynamical system described by time-based differential equations, including missiles. By providing this template-like structure, the code describing the mathematical formulation of the models is cleanly segregated from the code that comprises the underlying simulation engine.

The separation of model control functions from the models themselves also has another benefit: the code itself serves as a concise specification of the system being simulated. The model pseudo-code in *Figure A-4* clearly documents the model simulated without the distracting presence of extraneous code for lower-level simulation control functions.
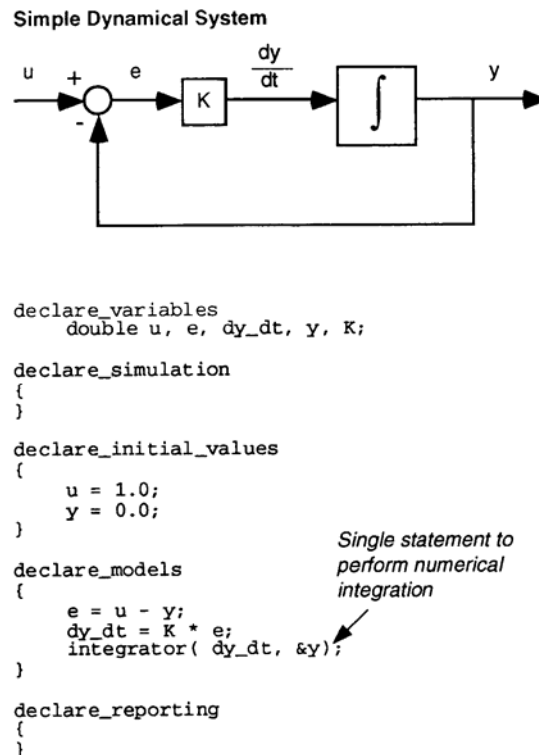
Figure A-4. Pseudo-Code Representation of a Simple Linear Dynamical System

A corollary to the idea of functional segregation is that the code should be separated as cleanly as possible into parts that are and are not modified by the user. The analyst who is writing models and incorporating models into the simulation should be abstracted by means of high-level functions from the parts of the code having to do with control. For example, the modeler is generally not concerned with the internal calculations of a numerical integrator; all that is required is that the integrator be available to perform integration as needed. Thus the analyst should not be required to manipulate (or even see) the code that actually performs the integration. Instead a high-level function should be available that provides access to the integration function.

4.2 Transparent Numerical Integration

4.2.1 Issue

Discrete time-stepping through the state evolution of a continuous dynamic system in an integration time loop is purely an artifact of simulating these systems on digital computer and has no basis in physical reality. It is doubtful that somewhere in the cosmos a master clock is governing every movement in the universe with a fourth-order RK integrator. How many times have simulation analyses been delayed because "We're having problems with the integrator." Why should the modeler, who is trying to simulate reality, be concerned with this artifact that is clearly not real in the first place?

4.2.2 Requirement

Performing numerical integration in a simulation should be as easy as calling a function, much like the cosine function returns the cosine of an angle. Using the differential equation solving structure shown in *Figure A-2*, integration should be accomplished with a single integrator statement, similar to the pseudo-code in *Figure A-4*.

4.3 Ability to Easily Mix Discrete and Continuous Models

4.3.1 Issue

The advent of digital avionics virtually ensures that any missile simulation will have to provide for sampled data processes in parallel with integrating continuous differential equations. Sampled processes during missile flight include the digital computations of the onboard flight computer as well as one-time events such as stage separation, seeker startup, and external navigation updates.

The problem is merging the timing for these sampled events with the numerical integration algorithm's time-step. This may sound trivial if the integrating time-step is hard-wired to 0.01 second and the flight computer issues updates every 0.1 second. However, the logic to merge discrete operations with the integrating time-step can become unwieldy with the complications of multiple-passes-per-time-step integration algorithms, multiple data rates from different processes, asynchronous data rates, and events that can occur at any time (not necessarily at a sample time). Requiring the analyst to write code to contend with this complexity is clearly undesirable.

4.3.2 Requirement

In line with the philosophy of shielding the developer from unnecessary complexity, a sample function should be provided in the architecture. *Figure A-5* illustrates use of the sample function to add discrete sampling to the simple linear dynamic system previously shown in *Figure A-4*. Adding a digital component is no more complex than enclosing the discrete time functions inside a conditional test of the sample function. The underlying simulation engine should take care of ensuring the enclosed computations are calculated at the correct time and that the continuous integration is correctly performed.

Simple System with Digital Component
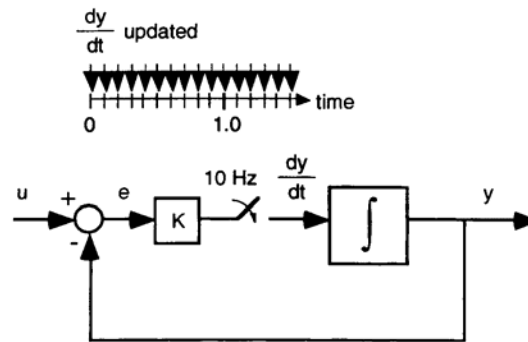

```
declare_variables
     double u, e, dy_d , y, K;

declare_simulation
{
}

declare_initial_values
{
     u = 1.0;
     y = 0.0;
}

declare_models                    Gets executed
{                                 every 0.1 sec
     if( sample( 0.1, 0.0)
     {
          e = u - y;
     }
     dy_dt = K * e;
     integrator(dy_dt,&y);
}

declare_reporting
{
}
```

Figure A-5. Adding a Digital Component to the Linear System

4.4 Get Data Into the Simulation Easily

4.4.1 Issue

The bane of all simulation developers since the advent of digital computers has been getting data into the simulation in the first place. The Graphical-User-Interface (GUI) is an attempt to address this problem. Certainly any effort expended on the part of the simulation developer to write code that reads data into the simulation is time not spent modeling or doing analysis. How much time is wasted precisely configuring input data files simply to get the simulation to run? How many times have simulations produced erroneous results (many time unnoticed) because input data was not correctly formatted?

4.4.2 Requirement

The full issue of user interfaces to communicate data to a simulation is certainly much too large in scope to attempt to address with any single simulation example or methodology. Also, recognizing that GUIs are highly dependent on the platform and operating system, they are certainly not as portable as standard C++ source code. Recognizing this, a reasonable approach is to narrow the possibilities considerably by examining how a simple text input file could be configured to expedite the data input process.

While a text input file can never be made as user-friendly as a well thought-out windows-based GUI, several disadvantages of text-based input files could be remedied without resorting to a GUI. Undesirable aspects of text files include that the data must often be precisely formatted in some manner and it must appear in the same order as the statements that read it in the source code. Further contemplation of these disadvantages reveals that it is not the text file format itself that enforces these constraints but how the input code is written.

The simulation should provide simple APIs for getting the data into the simulation with a text file. The APIs should allow for as much variation as possible as to how the data appears in the file. This variation includes that the data can appear in any order, with no format restrictions, and with explanatory comments embedded. Data should be quickly accessed with simple read commands and the underlying mechanics completely hidden from the simulation developer. Given a windows-based, mouse-driven text editor, the efficiency of this manner of input should be competitive with a full-fledged GUI and, in some ways, better. No system specific capabilities should be used to implement the data-access APIs.

4.5 Reliable and Efficient Manipulation of Tabular Data

4.5.1 Issue

Most, if not all, simulations use tables to describe the parameters of a process characterized by test or physical measurement. The tables can be very simple ranging from a one-dimensional table for an atmosphere model of pressure versus altitude to very complex such as a three-dimensional table (or even higher) characterizing the missile's aerodynamics. As with input, any code the simulation developer writes to manipulate tabular data diverts attention from building the simulation models. A particular pitfall with manipulating tabular data is not only the sheer bulk of code for the low-level routines to manipulate the data (incrementing, nested loops, interpolation, etc.) but allocating space to store the data. Allocating space often presents an insidious problem in that these errors are often not detected by the compiler and the simulation appears to run just fine (giving incorrect answers, of course). Examples include overwriting memory or exceeding storage bounds.

4.5.2 Requirement

The simulation should provide a specialized table data type and associated functions for handling tables. All tables, whether they are one-, two-, or three-dimensional should be declared as a "table" type . Simple functions to read and perform table look-ups on this type should be provided. Storage allocation should be transparent relieving the developer of the error-prone burden of allocating space and keeping up with table sizes. Consistent with the flexible input system described earlier, there should be few restrictions on how or where the table appears in the data file.

INITIAL DISTRIBUTION LIST

<u>Copies</u>

Defense Technical Information Center   1
8725 John J. Kingman Road, Suite 0944
Fort Belvoir, VA 22060-6218

IIT Research Institute   1
ATTN: GACIAC
10 W. 35th Street
Chicago, IL. 60616

AMSRD-AMR   (Electronically Distributed)

AMSRD-AMR-SG,	Mr. George W. Snyder	1
	Ms. Loretta Painter	1
	Mr. Jeffery W. Hester	5
AMSRD-AMR-SS,	Mr. George A. Sanders III	3
AMSRD-IN-IC		2
AMSRD-L-G-I,	Mr. Dayn Beam	1

DESE Research, Inc.
ATTN: Mr. Ray Sells   5
       Mr. Michael Fennell   1
315 Wynn Drive
Huntsville, AL 35805